Progress

Designing Performance-Optimized JDBC® Applications

Introduction

Recognized as experts in database access standards such as ODBC, Java® JDBC®, and ADO/OLE DB, Progress has consistently been instrumental in the development of new database access standards specifications. DataDirect was the first vendor to ship commercial JDBC drivers and the first vendor to be J2EE 1.3 and 1.4 JDBC certified across all major databases. Our role in helping Sun Microsystems, Inc write the original JDBC standard and subsequent modifications gives us unique technical insight into JDBC application performance.

Developing performance-oriented JDBC applications is not easy. JDBC drivers do not throw exceptions to tell you when your code is running too slow. This document presents some general guidelines for improving JDBC application performance that have been compiled by examining the JDBC implementations of numerous shipping JDBC applications. These guidelines include:

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

Following these guidelines can help you solve some common JDBC system performance problems, such as those listed in the following table.

Problem	Solution	See Guidelines in...
Network communication is slow.	Reduce network traffic.	"Using Database Metadata Methods" on page 2
Evaluation of complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	"Using Database Metadata Methods" on page 2 "Selecting JDBC Objects and Methods" on page 6
Excessive calls from the application to the driver slow performance.	Optimize application-to-driver interaction	"Retrieving Data" on page 4 "Selecting JDBC Objects and Methods" on page 6
Disk input/output is slow.	Limit disk input/output.	"Managing Connections and Updates" on page 10

In addition, most JDBC drivers provide options that improve performance, often with a tradeoff in functionality. If your application is not affected by functionality that is modified by setting a particular option, significant performance improvements can be realized. For details about performance-related options, refer to your JDBC driver documentation.

Using Database Metadata Methods

Because database metadata methods that generate ResultSet objects are slow compared to other JDBC methods, their frequent use can impair system performance. The guidelines in this section will help you optimize system performance when selecting and using database metadata.

Minimizing the Use of Database Metadata Methods

Compared to other JDBC methods, database metadata methods that generate ResultSet objects are relatively slow. Applications should cache information returned from result sets that generate database metadata methods so that multiple executions are not needed.

Although almost no JDBC application can be written without database metadata methods, you can improve system performance by minimizing their use. To return all result column information mandated by the JDBC specification, a JDBC driver may have to perform complex or multiple queries to return the necessary result set for a single call to a database metadata method. These particular elements of the SQL language are performance expensive.

Applications should cache information from database metadata methods. For example, call getTypeInfo 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 database metadata method, so the cache of information should not be difficult to maintain.

Avoiding Search Patterns

Using null arguments or search patterns in database metadata methods results in generating time-consuming queries. In addition, network traffic potentially increases due to unwanted results. Always supply as many non-null arguments as possible to result sets that generate database metadata methods.

Because database metadata methods are slow, invoke them in your applications as efficiently as possible. Many applications pass the fewest non-null arguments necessary for the function to return success. For example:

```
ResultSet WSrs = WSdbmd.getTables (null, null, "WSTable", null);
```

In this example, an application uses the getTables method to determine if the WSTable table exists. A JDBC driver interprets the request as: return all tables,

views, system tables, synonyms, temporary tables, and aliases named "WSTable" that exist in any database schema inside the database catalog.
In contrast, the following request provides non-null arguments as shown:

```
String[] tableTypes = {"TABLE"}; WSdbmd.getTables ("cat1",
      "johng", "WSTable", "tableTypes");
```

Clearly, a JDBC driver can process the second request more efficiently that it can process the first request.

Sometimes, little information is known about the object for which you are requesting information. Any information that the application can send the driver when calling database metadata methods can result in improved performance and reliability.

Using a Dummy Query to Determine Table Characteristics
Avoid using the getColumns method to determine characteristics about a table.
Instead, use a dummy query with getMetadata. Consider an application that allows the user to choose the columns that will be selected. Should the application use getColumns to return information about the columns to the user or, instead, prepare a dummy query and call getMetadata?

Case 1: getColumns Method

```
ResultSet WSrc = WSc.getColumns (... "UnknownTable" ...);
// This call to getColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
. . .
WSrc.next();
string Cname = getString(4);
. . .
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: getMetaData Method

```
// prepare dummy query
PreparedStatement WSps = WSc.prepareStatement
    ("SELECT * FROM UnknownTable WHERE 1 = 0");
// query is never executed on the server - only prepared ResultSetMetaData
WSsmd=WSps.getMetaData();
int numcols = WSrsmd.getColumnCount();
...
int ctype = WSrsmd.getColumnType(n)
...
// result column information has now been obtained
```

In both cases, a query is sent to the server. However, in Case 1, the potentially complex query must be prepared and executed, result description information must be formulated, and a result set of rows must be sent to the client. In Case 2, we prepare a simple query where we only retrieve result set information. Clearly, Case 2 is the better performing model.

To somewhat complicate this discussion, let us consider a DBMS server that does not natively support preparing a SQL statement. The performance of Case 1 does not change, but the performance of Case 2 improves slightly because the dummy query must be evaluated in addition to being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. For this situation, Case 2 still outperforms Case 1.

In summary, always use result set metadata to retrieve table column information, such as column names, column data types, and column precision and scale. Only use the getColumns method when the requested information cannot be obtained from result set metadata (for example, using the table column default values).

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 optimize system performance when retrieving data with JDBC applications.

Retrieving Long Data

Because retrieving long data across a network is slow and resource intensive, applications should not request long data unless it is necessary.

Most users don't want to see long data. If the user does want to see these result items, the application can query the database again, specifying only the 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 JDBC driver (that is, some applications `SELECT * FROM table_name ...`). If the Select list contains long data, most JDBC drivers are forced to retrieve that long data at fetch time, even if the application does not ask for the long data in the result set. When possible, the developer should attempt to implement a method that does not retrieve all columns of the table.

For example, consider the following code:

```
ResultSet rs = stmt.executeQuery (
   "SELECT * FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Remember that a JDBC driver cannot interpret an application's final intention. When a query is executed, the driver has no way to know which result columns an application will use. A driver anticipates that an application can request any of the result columns that are retrieved. When the JDBC driver processes the rs.next request, it will probably return at least one, if not more, result rows from the database server across the network. In this case, a result row contains all the column values for each row, including an employee photograph if the Employees table contains such a column. If you limit the Select list to contain only the employee name column, it results in decreased network traffic and a faster performing query at runtime.

For example:

```
ResultSet rs = stmt.executeQuery (
   "SELECT name FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Additionally, although the getClob and getBlob methods allow the application to control how long data is retrieved in the application, the developer must realize that in many cases, the JDBC driver emulates these methods due to the lack of true Large Object (LOB) locator support in the DBMS. In such cases, the driver must retrieve all the long data across the network before exposing the getClob and getBlob methods.

Reducing the Size of Data Retrieved

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.

To reduce network traffic and improve performance, you can reduce the size of any data being retrieved to some manageable limit by calling setMaxRows, setMaxFieldSize, and the driver-specific setFetchSize. Another method of reducing the size of the data being retrieved is to decrease the column size.

In addition, be careful to return only the rows and columns you need. If you return five columns when you only need two columns, performance is decreased, especially if the unnecessary rows include long data.

Choosing the Right Data Type

Retrieving and sending certain data types can be expensive. When you design a 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 database wire protocol.

Retrieving Result Sets

Most JDBC drivers cannot implement scrollable cursors because of limited support for scrollable cursors in the database system. Unless you are certain that the database supports using a scrollable result set, rs, for example, do not call rs.last and rs.getRow methods to find out how many rows the result set contains. If the JDBC driver emulates scrollable cursors, calling rs.last results in the driver retrieving all results across the network to reach the last row. Instead, you can either count the rows by iterating through the result set or get the number of rows by submitting a query with a COUNT column in the Select clause.

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

Selecting JDBC Objects and Methods

The guidelines in this section will help you select which JDBC objects and methods will give you the best performance.

Using Parameter Markers as Arguments to Stored Procedures

When calling stored procedures, always use parameter markers for argument markers instead of using literal arguments. JDBC drivers can call stored procedures on the database server either by executing the procedure as a SQL query or by optimizing the execution by invoking a Remote Procedure Call (RPC) directly on the database server. When you execute a stored procedure as a SQL query, the database server parses the statement, validates the argument types, and converts the arguments into the correct data types.

Remember that SQL is always sent to the database server as a character string, for example, "{call getCustName (12345)}". In this case, even though the application programmer may have assumed that the only argument to getCustName was an integer, the argument is actually passed inside a character string to the server. The database server parses the SQL query, isolates the single argument value 12345, and then, converts the string '12345' into an integer value before executing the procedure as a SQL language event.

By invoking a RPC on the database server, the overhead of using a SQL character string is avoided. Instead, the JDBC driver constructs a network packet that contains the parameters in their native data type formats and executes the procedure remotely.

Case 1: Not Using a Server-Side RPC

In this example, the stored procedure getCustName cannot be optimized to use a server-sideRPC. The database server must treat the SQL request as a normal language event, which includes parsing the statement, validating the argument types, and converting the arguments into the correct data types before executing the procedure.

```
CallableStatement cstmt = conn.prepareCall (
   "{call getCustName (12345)}");
ResultSet rs = cstmt.executeQuery ();
```

Case 2: Using a Server-Side RPC

In this example, the stored procedure getCustName can be optimized to use a server-side RPC. Because the application avoids literal arguments and calls the procedure by specifying all arguments as parameters, the JDBC driver can optimize the execution by invoking the stored procedure directly on the database as an RPC. The SQL language processing on the database server is avoided and execution time is greatly improved..

```
CallableStatement cstmt – conn.prepareCall (
   "{call getCustName (?)}");
cstmt.setLong (1,12345);
ResultSet rs = cstmt.executeQuery();
```

Using the Statement Object Instead of the PreparedStatement Object

JDBC drivers are optimized based on the perceived use of the functions that are being executed. Choose between the PreparedStatement object and the Statement object depending on how you plan to use the object. The Statement object is optimized for a single execution of a SQL statement. In contrast, the PreparedStatement object is optimized for SQL statements that will be executed two or more times.

The overhead for the initial execution of a PreparedStatement object is high. The benefit comes with subsequent executions of the SQL statement. For example, suppose we are preparing and executing a query that returns employee information based on an ID. Using a PreparedStatement object, a JDBC driver would process the prepare request by making a network request to the database server to parse and optimize the query. The execute results in another network request. If the application will only make this request once during its lifespan, using a Statement object instead of a PreparedStatement object results in only a single network roundtrip to the database server. Reducing network communication typically provides the most performance gains.

This guideline is complicated by the use of prepared statement pooling because the scope of execution is longer. When using prepared statement pooling, if a query only will be executed once, use the Statement object. If a query will be executed infrequently, but may be executed again during the life of a statement pool inside a connection pool, use a Prepared Statement object. Under similar circumstances without statement pooling, use the Statement object.

Using Batches Instead of Prepared Statements

Updating large amounts of data typically is done by preparing an Insert statement and executing that statement multiple times, resulting in numerous network roundtrips. To reduce the number of JDBC calls and improve performance, you can send multiple queries to the database at a time using the addBatch method of the PreparedStatement object. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
PreparedStatement ps = conn.prepareStatement(
      "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {

     ps.setString(name[n]);
     ps.setLong(id[n]);
     ps.setInt(salary[n]);
     ps.executeUpdate();
     }
```

Case 2: Using a Batch

```
PreparedStatement ps = conn.prepareStatement(
      "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {

     ps.setString(name[n]);
     ps.setLong(id[n]);
     ps.setInt(salary[n]);
     ps.addBatch();

}
ps.executeBatch();
```

In Case 1, a prepared statement is used to execute an Insert statement multiple times. In this case, 101 network roundtrips are required to perform 100 Insert operations: one roundtrip to prepare the statement and 100 additional roundtrips to execute its iterations. When the addBatch method is used to consolidate 100 Insert operations, as demonstrated in Case 2, only two network roundtrips are required — one to prepare the statement and another to execute the batch. Although more database CPU cycles are involved by using batches, performance is gained through the reduction of network roundtrips. Remember that the biggest gain in performance is realized by reducing network communication between the JDBC driver and the database server.

Choosing the Right Cursor

Choosing the appropriate type of cursor allows maximum flexibility for your application. This section summarizes the performance issues of three types of cursors: forward-only, insensitive, and sensitive.

A forward-only cursor provides excellent performance for sequential reads of all rows in a table. For retrieving table data, there is no faster way to retrieve result rows than using a forward-only cursor. However, forward-only cursors cannot be used when the rows to return are not sequential.

Insensitive cursors used by JDBC drivers are ideal for applications that require high levels of concurrency on the database server and require the ability to scroll forwards and backwards through result sets. In most cases, the first request to an insensitive cursor fetches all the rows and stores them on the client. If a driver uses a "lazy" fetching (fetch-on-demand) implementation, the first request may include many rows, if not all rows. Therefore, the initial request is very slow, especially when long data is retrieved. Subsequent requests do not require any network traffic (or, when a driver uses "lazy" fetching, requires limited network traffic) and are processed quickly.

Because the first request is processed slowly, insensitive cursors should not be used for a single request of one row. Developers should also avoid using insensitive cursors when long data or large result sets are returned because memory can be exhausted. Some insensitive cursor implementations cache the data in a temporary table on the database server and avoid the performance issue, but most cache the information local to the application.

Sensitive cursors, sometimes called keyset-driven cursors, use identifiers, such as a ROWID, that already exist in your database. When you scroll through the result set, the data for these identifiers is retrieved. Because each request generates network traffic, performance can be very slow. However, returning non-sequential rows does not further affect performance.

To illustrate this point further, consider an application that would normally return 1000 rows to an application. At execute time, or when the first row is requested, a JDBC driver does not execute the Select statement that was provided by the application. Instead, the JDBC driver replaces the Select list of the query with the key identifier, for example, ROWID. This modified query is then executed by the driver and all 1000 key values are retrieved by the database server and cached for use by the driver. Each request from the application for a result row directs the JDBC driver to look up the key value for the appropriate row in its local cache, construct an optimized query that contains a Where clause similar to WHERE ROWID=?, execute the modified query, and retrieve the single result row from the server.

Sensitive cursors are the preferred scrollable cursor model for dynamic situations when the application cannot afford to buffer the data associated with an insensitive cursor.

Using get Methods Effictively

JDBC provides a variety of methods to retrieve data from a result set, such as getInt, getString, and getObject. The getObject method is the most generic and provides the worst performance when the non-default mappings are specified. This is because the JDBC driver must perform extra processing to determine the type of the value being retrieved and generate the appropriate mapping. Always use the specific method for the data type.

To further improve performance, provide the column number of the column being retrieved, for example, getString(1), getLong(2), and getInt(3), instead of the column name. If column numbers are not specified, network traffic is unaffected, but costly conversions and lookups increase. For example, suppose you use `getString("foo") ...` A driver might have to convert `foo` to uppercase (if necessary), and then compare `foo` with all the columns in the column list. If, instead, the driver went directly to result column 23, a significant amount of processing would be saved.

For example, suppose you have a result set that has 15 columns and 100 rows, and the column names are not included in the result set. You are interested in three columns, EMPLOYEENAME (a string), EMPLOYEENUMBER (a long integer), and SALARY (an integer). If you specify getString("EmployeeName"), getLong("EmployeeNumber"), and getInt("Salary"), each column name must be converted to the appropriate case of the columns in the database metadata and lookups would increase considerably. Performance would improve significantly if you specify getString(1), getLong(2), and getInt(15).

Retrieving Auto-Generated Keys

Many databases have hidden columns (called pseudo-columns) that represent a unique key over every row in a table. Typically, using these types of columns in a query is the fastest way to access a row because the pseudo-columns usually represent the physical disk address of the data. Prior to JDBC 3.0, an application could only retrieve the value of the pseudo-columns by executing a Select statement immediately after inserting the data.

For example:

```
//insert row
int rowcount = stmt.executeUpdate (
   "INSERT INTO LocalGeniusList (name) values ('Karen')");
// now get the disk address – rowid – for the newly inserted row
ResultSet rs = stmt.executeQuery (
      "SELECT rowid FROM LocalGeniusList WHERE name = 'Karen'");
```

Retrieving pseudo-columns this way has two major flaws. First, retrieving the pseudo-column requires a separate query to be sent over the network and executed on the server. Second, because there may not be a primary key over the table, the search condition of the query may be unable to uniquely identify the row. In the latter case, multiple pseudo-column values can be returned, and the application may not be able to determine which value is actually the value for the most recently inserted row.

An optional feature of the JDBC 3.0 specification is the ability to retrieve auto-generated key information for a row when the row is inserted into a table. For example:

```
int rowcount = stmt.executeUpdate (
   "INSERT INTO LocalGeniusList (name) VALUES ('Karen')",
// insert row AND return key
Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys ();
// key is automatically available
```

Now, the application contains a value that can be used in a search condition to provide the fastest access to the row and a value that uniquely identifies the row, even when a primary key doesn't exist on the table.

The ability to retrieve auto-generated keys provides flexibility to the JDBC developer and creates performance boosts when accessing data.

Managing Connections and Updates

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

Managing Connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement objects, 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. For example, some applications establish a connection and then call a method in a separate component that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the established connection object 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 perform SQL statements. Connection objects can have multiple statement objects associated with them. Statement objects, which are defined to be memory storage for information about SQL statements, can manage multiple SQL statements.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web. Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network input/output needed to create a new connection.

In addition to connection pooling tuning options, JDBC 3.0 also specifies semantics for providing a statement pool. Similar to connection pooling, a statement pool caches PreparedStatement objects so that they can be re-used from a cache without application intervention. For example, an application may create a PreparedStatement object similar to the following SQL statement:

```
SELECT name, address, dept, salary FROM personnel
WHERE empid = ? or name = ? or address = ?"
```

When the PreparedStatement object is created, the SQL query is parsed for semantic validation and a query optimization plan is produced. The process of creating a prepared statement is extremely expensive in terms of performance with some database systems, such as DB2. Once the prepared statement is closed, a JDBC 3.0-compliant driver places the prepared statement into a local cache instead of discarding it. If the application later attempts to create a prepared statement with the same SQL query, a common occurrence in many applications, the driver can simply retrieve the associated statement from the local cache instead of performing a network roundtrip to the server and an expensive database validation.

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

Making Commits in Transactions

Committing transactions is slow because of the amount of disk input/output, and potentially network input/output, that is required. Always turn Autocommit off by using the WSConnection.setAutoCommit(false) setting.

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 usually a sequential write to a journal file, but nevertheless, it involves disk input/output. By default, Autocommit is on when connecting to a data source, and Autocommit mode usually impairs performance because of the significant amount of disk input/output needed to commit every operation.

Furthermore, most database servers do not provide a native Autocommit mode. For this type of server, the JDBC 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, 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 longer than necessary, 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 (the JDBC driver, the transaction monitor, and the DBMS). Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible. Many Java application servers provide a default transaction behavior that uses distributed transactions.

For the best system performance, design the application to run using a single Connection object.

Using updateXXX Methods

Although programmatic updates do not apply to all types of applications, developers should attempt to use programmatic updates and deletes. Using the updateXXX methods of the ResultSet object allows the developer to update data without building a complex SQL statement. Instead, the developer simply supplies the column in the result set that is to be updated and the data that is to be changed. Then, before moving the cursor from the row in the result set, the updateRow method must be called to update the database as well.

In the following code fragment, the value of the Age column of the ResultSet object rs is retrieved using the method getInt, and the method updateInt is used to update the column with an int value of 25. The method updateRow is called to update the row in the database that will contain the modified value.

```
int n = rs.getInt("Age");
// n contains value of Age column in the resultset rs
...
rs.updateInt("Age", 25);
rs.updateRow();
```

In addition to making the application more easily maintainable, programmatic 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 getBestRowIdenifier

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

Some applications cannot be designed to take advantage of positional updates and deletes. Some applications might formulate the Where clause by using all searchable result columns, by calling getPrimaryKeys, or by calling getIndexInfo to find columns that might be part of a unique index. These methods usually work, but might result in fairly complex queries.

Consider the following example:

```
ResultSet WSrs = WSs.executeQuery
   ("SELECT first_name, last_name, ssn, address, city, state, zip     FROM
emp");
// fetch data
...
WSs.executeUpdate ("UPDATE emp SET address = ?
WHERE first_name = ? and last_name = ? and ssn = ?     and address = ? and
city = ?
and state = ?
   and zip = ?");
// fairly complex query
```

Applications should call getBestRowIdentifier to retrieve the optimal set of columns (possibly a pseudo-column) that identifies a specific 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 generally provide the fastest access to the data because they typically are pointers to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from getColumns. To determine if pseudo-columns exist, call getBestRowIdentifier.

Consider the previous example again:

```
...
ResultSet WSrowid = getBestRowIdentifier()
   (... "emp", ...);
...
WSs.executeUpdate ("UPDATE emp SET address = ?
     WHERE ROWID = ?";
// fastest access to the data!
```

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

Conclusion

Although developing performance-oriented JDBC applications is not easy, you can improve system performance by following the general guidelines described in this document:

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

Progress was the first vendor to ship commercial JDBC drivers and the first vendor to be J2EE 1.3 and 1.4 JDBC certified across all major databases. Our role in helping Sun Microsystems, Inc write the original JDBC standard and subsequent modifications gives us unique technical insight into JDBC application performance. With thoughtful design and implementation, you can ensure that your JDBC applications run more efficiently and generate less network traffic.

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying mission-critical business applications. Progress empowers enterprises and ISVs to build and deliver cognitive-first applications that harness big data to derive business insights and competitive advantage. Progress offers leading technologies for easily building powerful user interfaces across any type of device, a reliable, scalable and secure backend platform to deploy modern applications, leading data connectivity to all sources, and award-winning predictive analytics that brings the power of machine learning to any organization. Over 1,700 independent software vendors, 100,000 enterprise customers, and two million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw
For regional international office locations and contact information,
please go to www.progress.com/worldwide

Rev 2018/03 | RITM0014390

Progress®