C

JDBC Coding Tips

This appendix describes methods to optimize a Java Database Connectivity (JDBC) application. It includes the following topics:

- JDBC and Multithreading
- Performance Optimization of JDBC Programs
- Transaction Isolation Levels and Access Modes in JDBC

C.1 JDBC and Multithreading

Oracle JDBC drivers provide full support for, and are highly optimized for, applications that use Java multithreading. Controlled serial access to a connection, such as that provided by connection caching, is both necessary and encouraged. However, Oracle strongly discourages sharing a database connection among multiple threads. Avoid allowing multiple threads to access a connection simultaneously. If multiple threads must share a connection, use a disciplined begin-using/end-using protocol.

Keep the following points in mind while working on multithreaded applications:

- Use the `Connection` object as a local variable.
- Close the connection in the `finally` block before exiting the method. For example:

```
Connection conn = null;
try
{
    ...
}
finally
{
   if(conn != null) conn.close();
}
```

- Do not share `Connection` objects between threads.
- Never synchronize on JDBC objects because it is done internally by the driver.
- Use the `Statement.setQueryTimeout` method to set the time to execute a query instead of cancelling the long-running query from a different thread.
- Use the `Statement.cancel` method for SQL operations like `SELECT`, `UPDATE`, or `DELETE`.
- Use the `Connection.cancel` method for SQL operations like `COMMIT`, `ROLLBACK`, and so on.
- Do not use the `Thread.interrupt` method.

C.2 Performance Optimization of JDBC Programs

You can significantly enhance the performance of your JDBC programs by using any of these features:

- Disabling Auto-Commit Mode

C.2.1 Disabling Auto-Commit Mode

Auto-commit mode indicates to the database whether to issue an automatic `COMMIT` operation after every SQL operation. Being in auto-commit mode can be expensive in terms of time and processing effort if, for example, you are repeating the same statement with different bind variables.

By default, new connection objects are in auto-commit mode. However, you can disable auto-commit mode with the `setAutoCommit` method of the connection object, either `java.sql.Conection` or `oracle.jdbc.OracleConnection`.

In auto-commit mode, the `COMMIT` operation occurs either when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a `ResultSet` object, the statement completes when the last row of the Result Set has been retrieved or when the Result Set has been closed. In more complex cases, a single statement can return multiple results as well as output parameter values. Here, the `COMMIT` occurs when all results and output parameter values have been retrieved.

If you disable auto-commit mode with a `setAutoCommit(false)` call, then you must manually commit or roll back groups of operations using the `commit` or `rollback` method of the connection object.

Example

The following example illustrates loading the driver and connecting to the database. Because new connections are in auto-commit mode by default, this example shows how to disable auto-commit. In the example, `conn` represents the `Connection` object, and `stmt` represents the `Statement` object.

```
// Connect to the database
// You can put a database host name after the @ sign in the connection URL.
   OracleDataSource ods = new OracleDataSource();
   ods.setURL("jdbc:oracle:oci:@");
   ods.setUser("HR");
   ods.setPassword("hr");
   Connection conn = ods.getConnection();

// It's faster when auto commit is off
conn.setAutoCommit (false);

// Create a Statement
Statement stmt = conn.createStatement ();
...
```

C.2.2 Standard Fetch Size and Oracle Row Prefetching

Oracle JDBC connection and statement objects allow you to specify the number of rows to prefetch into the client with each trip to the database, while a result set is being populated during a query.

You can set a value in a connection object that affects each statement produced through that connection, and you can override that value in any particular statement object. The default value in a connection object is 10. Prefetching data into the client reduces the number of round-trips to the server.

Similarly, and with more flexibility, JDBC 2.0 enables you to specify the number of rows to fetch with each trip, both for statement objects (affecting subsequent queries) and for result set objects (affecting row refetches). By default, a result set uses the value for the statement object that produced it. If you do not set the JDBC 2.0 fetch size, then the Oracle connection row-prefetch value is used by default.

Related Topics

- Row Fetch Size

C.2.2.1 Row Prefetch Auto Tuning

Starting from Oracle Database Release 23ai, the JDBC Thin driver has a new row prefetch auto-tuning feature.

As mentioned earlier, the default prefetch size is 10. If your application does not alter this default value, then prefetch auto tuning is activated after the fourth fetch because the auto tuning algorithm computes the new prefetch size value based on the average row size in the first three prefetch operations. If your application alters the default prefetch value, then the auto tuning feature is disabled and the prefetch size is set to the value that you specified.

The JDBC connection property `oracle.jdbc.fetchSizeTuning` is also used to control the fetch size tuning behavior. The default value of this property is 8. Setting the value to 0 disables the auto tuning feature. The computed or tuned row prefetch size by the auto tuning algorithm can vary between a minimum value of 4 and a maximum value of 250. The algorithm to compute the row prefetch size is transparent to the user.

> **Note:**
>
> The `statement.setFetchSize` method takes priority over the `oracle.jdbc.fetchSizeTuning` method because it disables auto tuning for that particular statement. If you set the fetch size explicitly on a statement, then the auto tuning for that particular statement is disabled. If you set the prefetch size using the `oracle.jdbc.defaultRowPrefetch` connection property, then it affects all statements created from that connection.

> **See Also:**
>
> Oracle Database JDBC Java API Reference

C.2.3 About Setting the Session Data Unit Size

Session data unit (SDU) is a buffer that Oracle Net uses to place data before transmitting it across the network. Oracle Net sends the data in the buffer either when the request is completed or when it is full.

You can configure the SDU and obtain the following benefits, among others:

- Reduction in the time required to transmit a SQL query and result across the network
- Transmission of larger chunks of data

> **Note:**
>
> The footprint of the client and the server process increase if you set a bigger SDU size.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide*

This section describes the following:

- About Setting the SDU Size for the Database Server
- About Setting the SDU Size for JDBC Thin Client

C.2.3.1 About Setting the SDU Size for the Database Server

To set the SDU size for the database server, configure the `DEFAULT_SDU_SIZE` parameter in the `sqlnet.ora` file.

C.2.3.2 About Setting the SDU Size for JDBC OCI Client

The JDBC OCI client uses Oracle Net layer. So, you can set the SDU size for the JDBC OCI client by configuring the `DEFAULT_SDU_SIZE` parameter in the `sqlnet.ora` file.

C.2.3.3 About Setting the SDU Size for JDBC Thin Client

You can set the SDU size for JDBC thin client by specifying it in the `DESCRIPTION` parameter for a particular connection descriptor.

```
sales.example.com=
(DESCRIPTION=
    (SDU=11280)
      (ADDRESS=(PROTOCOL=tcp)(HOST=sales-server)(PORT=5221))
    (CONNECT_DATA=
     (SERVICE_NAME=sales.example.com))
)
```

C.2.4 JDBC Update Batching

Oracle JDBC drivers enable you to accumulate `INSERT`, `DELETE`, and `UPDATE` operations of prepared statements at the client and send them to the server in batches. This feature reduces round-trips to the server.

> **Note:**
>
> Oracle recommends to keep the batch sizes in the range of 100 or less. Larger batches provide little or no performance improvement and may actually reduce performance due to the client resources required to handle the large batch.

C.2.5 Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Applications use the statement cache to cache statements associated with a particular physical connection. When you enable Statement caching, a Statement object is cached when you call the `close` method. Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections.

> **Note:**
>
> The Oracle JDBC drivers are optimized for use with the Oracle Statement cache. Oracle strongly recommends that you use the Oracle Statement cache (implicit or explicit).

When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

Related Topics

- Statement and Result Set Caching

C.2.6 Mapping Between Built-in SQL and Java Types

The SQL built-in types are those types with system-defined names, such as `NUMBER`, and `CHAR`, as opposed to the Oracle objects, varray, and nested table types, which have user-defined names. In JDBC programs that access data of built-in SQL types, all type conversions are unambiguous, because the program context determines the Java type to which a SQL datum will be converted.

Table C-1 Mapping of SQL Data Types to Java Classes that Represent SQL Data Types

SQL Data Type	ORACLE Mapping - Java Classes Representing SQL Data Types
`CHAR`	`oracle.sql.CHAR`

Table C-1 (Cont.) Mapping of SQL Data Types to Java Classes that Represent SQL Data Types

SQL Data Type	ORACLE Mapping - Java Classes Representing SQL Data Types
VARCHAR2	oracle.sql.CHAR
DATE	oracle.sql.DATE
DECIMAL	oracle.sql.NUMBER
DOUBLE PRECISION	oracle.sql.NUMBER
FLOAT	oracle.sql.NUMBER
INTEGER	oracle.sql.NUMBER
REAL	oracle.sql.NUMBER
RAW	oracle.sql.RAW
LONG RAW	oracle.sql.RAW
REF CURSOR	java.sql.ResultSet
CLOB LOCATOR	oracle.jdbc.OracleClob[1]
BLOB LOCATOR	oracle.jdbc.OracleBlob[2]
BFILE	oracle.sql.BFILE
nested table	oracle.jdbc.OracleArray[3]
varray	oracle.jdbc.OracleArray
SQL object value	If there is no entry for the object value in the type map: • oracle.jdbc.OracleStruct[4] If there is an entry for the object value in the type map: • customized Java class
REF to SQL object type	class that implements oracle.sql.SQLRef, typically by implementing oracle.jdbc.OracleRef[5]

1 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.CLOB class is deprecated and replaced with the oracle.jdbc.OracleClob interface.
2 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced with the oracle.jdbc.OracleBlob interface.
3 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface.
4 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface.
5 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.REF class is deprecated and replaced with the oracle.jdbc.OracleRef interface.

The most efficient way to access numeric data is to use primitive Java types like int, float, long, and double. However, the range of values of these types do not exactly match the range of values of the SQL NUMBER data type. As a result, there may be some loss of information. If absolute precision is required across the entire value range, then use the BigDecimal type.

All character data is converted to the UCS2 character set of Java. The most efficient way to access character data is as java.lang.String. In worst case, this can cause a loss of information when two or more characters in the database character set map to a single UCS2 character. Since Oracle Database 11*g*, all characters in the character set map to the characters in the UCS2 character set. However, some characters do map to surrogate pairs.

C.3 Transaction Isolation Levels and Access Modes in JDBC

Read-only connections are supported by Oracle JDBC drivers, but not by the Oracle server.

For transactions, the Oracle server supports only the `TRANSACTION_READ_COMMITTED` and `TRANSACTION_SERIALIZABLE` transaction isolation levels. The default is `TRANSACTION_READ_COMMITTED`. Use the following methods of the `oracle.jdbc.OracleConnection` interface to get and set the level:

- `getTransactionIsolation`: Gets the current transaction isolation level of the connection.
- `setTransactionIsolation`: Changes the transaction isolation level, using either the `TRANSACTION_READ_COMMITTED` or the `TRANSACTION_SERIALIZABLE` value.