E
Troubleshooting

This appendix describes how to troubleshoot a Java Database Connectivity (JDBC) application in the following topics:

- Common Problems
- Basic Debugging Procedures

E.1 Common Problems

This section describes some common problems that you might encounter while using Oracle JDBC drivers. These problems include:

- Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables
- Memory Leaks and Running Out of Cursors
- Opening More than 16 OCI Connections for a Process
- Using statement.cancel
- Using JDBC with Firewalls
- Frequent Abrupt Disconnection from Server
- Network Adapter Cannot Establish Connection

E.1.1 Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables

In PL/SQL, when a `CHAR` or a `VARCHAR2` column is defined as a `OUT` or `IN/OUT` variable, the driver allocates a `CHAR` array of 32512 chars. This can cause a memory consumption problem. JDBC Thin driver does not allocate memory when using `VARCHAR2` output type. But JDBC OCI driver allocates memory for both `CHAR` and `VARCHAR2` types. So, CPU load in OCI driver is higher than Thin driver.

At previous releases, the solution to the problem was to invoke the `Statement.setMaxFieldSize` method. A better solution is to use `OracleCallableStatement.registerOutParameter`. Oracle encourages you always to call `registerOutParameter (int paramIndex, int sqlType, int scale, int maxLength)` on each `CHAR` or `VARCHAR2` column. This method is defined in `oracle.jdbc.OracleCallableStatement`. Use the fourth argument, `maxLength`, to limit the memory consumption. This parameter tells the driver how many characters are necessary to store this column. The column is truncated if the character array cannot hold the column data. The third argument, `scale`, is ignored by the driver.

E.1.2 Memory Leaks and Running Out of Cursors

If you receive messages that you are running out of cursors or that you are running out of memory, make sure that all your `Statement` and `ResultSet` objects are explicitly closed. Oracle JDBC drivers do not have finalizer methods. They perform cleanup routines by using the `close`

method of the `ResultSet` and `Statement` classes. If you do not explicitly close your result set and statement objects, significant memory leaks can occur. You could also run out of cursors in the database. Closing a statement releases the corresponding cursor in the database.

Similarly, you must explicitly close `Connection` objects to avoid leaking and running out of cursors on the server-side. When you close the connection, the JDBC driver closes any open statement objects associated with it, thus releasing the cursor on the server-side.

E.1.3 Opening More than 16 OCI Connections for a Process

You may find that you are unable to open more than approximately 16 JDBC-OCI connections for a process at any given time. The most likely reasons for this would be either that the number of processes on the server exceeded the limit specified in the initialization file, or that the per-process file descriptors limit was exceeded. It is important to note that one JDBC-OCI connection can use more than one file descriptor (it might use anywhere between 3 and 4 file descriptors).

If the server allows more than 16 processes, then the problem could be with the per-process file descriptor limit. The possible solution would be to increase this limit.

E.1.4 Using statement.cancel

The JDBC standard method `Statement.cancel` attempts to cleanly stop the execution of a SQL statement by sending a message to the database. In response, the database stops execution and replies with an error message. The Java thread that invoked `Statement.execute` waits on the server, and continues execution only when it receives the error reply message invoked by the call of the other thread to `Statement.cancel` method.

As a result, the `Statement.cancel` method relies on the correct functioning of the network and the database. If either the network connection is broken or the database server is hung, the client does not receive the error reply to the cancel message. Frequently, when the server process dies, JDBC receives an `IOException` that frees the thread that invoked `Statement.execute`. In some circumstances, the server is hung, but JDBC does not receive an `IOException`. The `Statement.cancel` method does not free the thread that initiated the `Statement.execute` method.

> **Note:**
>
> Remember the following points while working with the `Statement.cancel` method:
>
> - Distinguish between Connection-level and Statement-level cancel. If a nonstatement execution, for example, a `ROLLBACK` is cancelled by a `statement.cancel` method, then we replay the command (only if it is `ROLLBACK`, `COMMIT`, `autoCommit ON`, `autoCommit OFF`, `VERSION`). To guarantee data integrity, we do not replay statement executions.
> - Synchronize statement execution and statement cancel, so that the execution does not return until the cancel call is sent to the Database. This provides a better chance for the executing statement to be cancelled.
> - Synchronize cancel calls, so that any new cancel request is ignored until the cancel in progress has completed the full protocol, that is, after the database receives an interrupt, act on it, and notify JDBC.

When JDBC does not receive an `IOException`, Oracle Net may eventually time out and close the connection. This causes an `IOException` and frees the thread. This process can take many minutes. For information about how to control this time-out, see the description of the `readTimeout` property for `OracleDatasource.setConnectionProperties`. You can also tune this time-out with certain Oracle Net settings.

The JDBC standard method `Statement.setQueryTimeout` relies on the `Statement.cancel` method. If execution continues longer than the specified time-out interval, then the monitor thread calls the `Statement.cancel` method. This is subject to all the same limitations described previously. As a result, there are cases when the time-out does not free the thread that invoked the `Statement.execute` method.

The length of time between execution and cancellation is not precise. This interval is no less than the specified time-out interval but can be several seconds longer. If the application has active threads running at high priority, then the interval can be arbitrarily longer. The monitor thread runs at high priority, but other high priority threads may keep it from running indefinitely. Note that the monitor thread is started only if there are statements executed with non zero time-out. There is only one monitor thread that monitors all Oracle JDBC statement execution.

> **Note:**
>
> The `Statement.cancel` method and the `Statement.setQueryTimeout` method are not supported in the server-side internal driver. The server-side internal driver runs in the single-threaded server process and the Oracle JVM implements Java threads within this single-threaded process. If the server-side internal driver is executing a SQL statement, then no Java thread can call the `Statement.cancel` method. This also applies to the Oracle JDBC monitor thread.

E.1.5 Using JDBC with Firewalls

Firewall timeout for idle-connections may sever a connection. This can cause JDBC applications to hang while waiting for a connection. You can perform one or more of the following actions to avoid connections from being severed due to firewall timeout:

- If you are using connection caching or connection pooling, then always set the inactivity timeout value on the connection cache to be shorter than the firewall idle timeout value.
- Pass `oracle.jdbc.ReadTimeout` as connection property to enable read timeout on socket. The timeout value is in milliseconds.
- For both JDBC OCI and JDBC Thin drivers, use net descriptor to connect to the database and specify the `ENABLE=BROKEN` parameter in the `DESCRIPTION` clause in the connect descriptor. Also, set a lower value for `TCP_KEEPALIVE_INTERVAL`.
- Enable Oracle Net DCD by setting `SQLNET.EXPIRE_TIME=1` in the `sqlnet.ora` file on the server-side.

E.1.6 Frequent Abrupt Disconnection from Server

If the network is not reliable, then it is difficult for a client to detect the frequent disconnections when the server is abruptly disconnected. By default, a client running on Linux takes 7200 seconds (2 hours) to sense the abrupt disconnections. This value is equal to the value of the `tcp_keepalive_time` property. If you want your application to detect the disconnections faster,

then you must set the value of the `tcp_keepalive_time`, `tcp_keepalive_interval`, and `tcp_keepalive_probes` properties to a lower value at the operating system level.

> **Note:**
>
> Setting a low value for the `tcp_keepalive_interval` property leads to frequent probe packets on the network, which can make the system slower. So, the value of this property should be set appropriately based on the system requirements.

Also, you must specify the `ENABLE=BROKEN` parameter in the `DESCRIPTION` clause in the connection descriptor. For example:

```
jdbc:oracle:thin:@(DESCRIPTION=(ENABLE=BROKEN)(ADDRESS=(PROTOCOL=tcp)(PORT=5221)
(HOST=myhost))(CONNECT_DATA=(SERVICE_NAME=orcl)))
```

E.1.7 Network Adapter Cannot Establish Connection

You may receive the following error while trying to establish a connection from a JDBC application to an Oracle instance:

```
java.sql.SQLException: Io exception:
  The Network Adapter could not establish connection

SQLException: SQLState (null) vendor code (17002)
```

This error may occur even if all or any of the following conditions is true:

- You are able to establish a SQL*Plus connection from the same client to the same Oracle instance.
- You are able to establish a JDBC OCI connection, but not a JDBC Thin connection from the same client to the same Oracle instance.
- The same JDBC application is able to connect from a different client to the same Oracle instance.
- The same behavior applies whether the initial JDBC connection string specifies a host name or an IP address.

One or more of the following reasons can cause this error:

- The host name to which you are trying to establish the connection is incorrect.
- The port number you are using to establish the connection is wrong.
- The NIC card supports both IPv4 and IPv6.
- The Oracle instance is configured for MTS, but the JDBC connection uses a shared server instead of a dedicated server.

You can quickly diagnose these above-mentioned reasons by using SQL*Plus, except for the issue with the NIC card. The following sections specify how to resolve this error, and also contains a sample application:

- Oracle Instance Configured with MTS Server Uses Shared Server
- JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6
- Sample Application

E.1.7.1 Oracle Instance Configured with MTS Server Uses Shared Server

For resolving this error, you must verify whether the Oracle instance is configured for Multi-threaded Server (MTS) or not. If the Oracle instance is not configured for MTS, then it must be configured.

If the Oracle instance is configured for MTS, then you must force the JDBC connection to use a dedicated server instead of a shared server. You can achieve this by reconfiguring the server to use dedicated connections only. If it is not feasible to configure your server to use only dedicated connections, then you perform the following steps to set it from the client side:

For JDBC OCI Client

1. Add the `(SERVER=DEDICATED)` property to the TNS connection string stored in the `tnsnames.ora` file on the client.
2. Set the `USER_DEDICATED_SERVER=ON` in the `sqlnet.ora` file on the client.

For JDBC Thin:

You must specify a full name-value pair connection string (the same as it may appear in the `tnsnames.ora` file) instead of the short JDBC Thin syntax. For example, instead of the "`jdbc:oracle:thin:@host:port:sid`" connection string, you must use a connection string of the following form:

```
"jdbc:oracle:thin:@(DESCRIPTION="                    +
                    "(ADDRESS_LIST="                 +
                        "(ADDRESS=(PROTOCOL=TCP)"    +
                                 "(HOST=host)"       +
                                 "(PORT=port)"       +
                        ")"                          +
                    ")"                              +
                    "(CONNECT_DATA="                 +
                        "(SERVICE_NAME=sid)"         +
                        "(SERVER=DEDICATED)"         +
                    ")"                              +
                  ")"
```

E.1.7.2 JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6

If the Network Interface Controller (NIC) card of the server is configured to support both IPv4 and IPv6, then some services may start with IPv6. Any client application that tries to connect using IPv4 to the service that is running with IPv6 (or the other way round) receives a connection refused error. If a JDBC thin client application tries to connect to the Database server, then the application may stop responding or fail with the following error:

```
java.sql.SQLException: Io exception: The Network Adapter could not establish the
connection Error Code: 17002
```

Use any of the following solutions to resolve this error:

- Indicate the Java Virtual Machine (JVM) to use IP protocol version 4. Launch the JVM, where the JDBC application is running, with the `-Djava.net.preferIPv4Stack` parameter as `true`. For example, suppose you are running a JDBC application named `jdbcTest`. Then execute the application in the following way:

  ```
  java -Djava.net.preferIPv4Stack=true jdbcTest
  ```

- Use the OCI JDBC driver.

E.1.7.3 Sample Application

Example E-1 shows a basic JDBC program that connects to a Database and can be used to test your connection. It enables to try all forms of connection using Oracle JDBC drivers.

Example E-1 Basic JDBC Program to Connect to a Database in Five Different Ways

```
import java.sql.*;
public class Jdbctest
{
   public static void main (String args[])
   {
         try
         {
                  /* Uncomment the next line for more connection information */
                  // DriverManager.setLogStream(System.out);
                  /* Set the host, port, and sid below to match the entries in the
listener.ora */
                  String host = "myhost.oracle.com";
                  String port = "5221";
                  String sid  = "orcl";
                  // or pass on command line arguments for all three items
                  if ( args.length >= 3 )
                  {
                            host = args[0];
                            port = args[1];
                            sid  = args[2];
                  }

                  String s1 = "jdbc:oracle:thin:@" + host + ":" + port + ":" + sid ;
                  if ( args.length == 1 )
                  {
                            s1 =  "jdbc:oracle:oci8:@" + args[0];
                  }
                  if ( args.length == 4 )
                  {
                              s1 = "jdbc:oracle:" + args[3] + ":@" +
                                   "(description=(address=(host=" + host+ ")
(protocol=tcp)(port=" + port+ "))(connect_data=(sid="+ sid + ")))";
                  }
                  System.out.println( "Connecting with: " );
                  System.out.println( s1 );
                  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                  Connection conn =
DriverManager.getConnection( s1,"hr","hr");
                  DatabaseMetaData dmd = conn.getMetaData();
                  System.out.println("DriverVersion:["+dmd.getDriverVersion()+"]");
                  System.out.println("DriverMajorVersion: ["+dmd.getDriverMajorVersion()
+"]");
                  System.out.println("DriverMinorVersion: ["+dmd.getDriverMinorVersion()
+"]");
                  System.out.println("DriverName:["+dmd.getDriverName()+"]");
                  if ( conn!=null )
                     conn.close();
                  System.out.println("Done.");
           }
           catch ( SQLException e )
           {
                  System.out.println ("\n*** Java Stack Trace ***\n");
                  e.printStackTrace();
                  System.out.println ("\n*** SQLException caught ***\n");
```

```
                    while ( e != null )
                    {
                                System.out.println ("SQLState: " + e.getSQLState
());
                                System.out.println ("Message:  " + e.getMessage
());
                                System.out.println ("Error Code:   " + e.getErrorCode
());
                                e = e.getNextException ();
                                System.out.println ("");
                     }
               }
  }
}
```

E.2 Basic Debugging Procedures

This section describes strategies for debugging a JDBC program:

- Oracle Net Tracing to Trap Network Events
- Third Party Debugging Tools

Related Topics

- About Processing SQL Exceptions

E.2.1 Oracle Net Tracing to Trap Network Events

You can enable client and server Oracle-Net trace to trap the packets sent over Oracle Net. You can use client-side tracing only for the JDBC OCI driver; it is not supported for the JDBC Thin driver.

The trace facility produces a detailed sequence of statements that describe network events as they execute. "Tracing" an operation lets you obtain more information about the internal operations of the event. This information is printed to a readable file that identifies the events that led to the error. Several Oracle Net parameters in the `SQLNET.ORA` file control the gathering of trace information. After setting the parameters in `SQLNET.ORA`, you must make a new connection for tracing to be performed.

The higher the trace level, the more detail is captured in the trace file. Because the trace file can be hard to understand, start with a trace level of 4 when enabling tracing. The first part of the trace file contains connection handshake information, so look beyond this for the SQL statements and error messages related to your JDBC program.

> **Note:**
>
> The trace facility uses a large amount of disk space and might have significant impact upon system performance. Therefore, enable tracing only when necessary.

Related Topics

- *Oracle Call Interface Programmer's Guide*

E.2.1.1 Client-Side Tracing

Set the following parameters in the `SQLNET.ORA` file on the client system.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported.

E.2.1.1.1 TRACE_LEVEL_CLIENT

Purpose:

Turns tracing `on` or `off` to a certain specified level.

Default Value:

0 or OFF

Available Values:

- 0 or `OFF` - No trace output
- 4 or `USER` - User trace information
- 10 or `ADMIN` - Administration trace information
- 16 or `SUPPORT` - WorldWide Customer Support trace information

Example:

```
TRACE_LEVEL_CLIENT=10
```

E.2.1.1.2 TRACE_DIRECTORY_CLIENT

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME`/network/trace`

Example:

UNIX: `TRACE_DIRECTORY_CLIENT=/oracle/traces`

Windows: `TRACE_DIRECTORY_CLIENT=C:\ORACLE\TRACES`

E.2.1.1.3 TRACE_FILE_CLIENT

Purpose:

Specifies the name of the client trace file.

Default Value:

`SQLNET.TRC`

Example:

```
TRACE_FILE_CLIENT=cli_Connection1.trc
```

> **Note:**
>
> Ensure that the name you choose for the `TRACE_FILE_CLIENT` file is different from the name you choose for the `TRACE_FILE_SERVER` file.

E.2.1.1.4 TRACE_UNIQUE_CLIENT

Purpose:

Gives each client-side trace a unique name to prevent each trace file from being overwritten with the next occurrence of a client trace. The `PID` is attached to the end of the file name.

Default Value:

`OFF`

Example:

```
TRACE_UNIQUE_CLIENT = ON
```

E.2.1.2 Server-Side Tracing

Set the following parameters in the `SQLNET.ORA` file on the server system. Each connection will generate a separate file with a unique file name.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported.

E.2.1.2.1 TRACE_LEVEL_SERVER

Purpose:

Turns tracing `on` or `off` to a certain specified level.

Default Value:

0 or `OFF`

Available Values:

- 0 or `OFF` - No trace output

- 4 or `USER` - User trace information
- 10 or `ADMIN` - Administration trace information
- 16 or `SUPPORT` - WorldWide Customer Support trace information

Example:

```
TRACE_LEVEL_SERVER=10
```

E.2.1.2.2 TRACE_DIRECTORY_SERVER

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME`/network/trace`

Example:

```
TRACE_DIRECTORY_SERVER=/oracle/traces
```

E.2.1.2.3 TRACE_FILE_SERVER

Purpose:

Specifies the name of the server trace file.

Default Value:

```
SERVER.TRC
```

Example:

```
TRACE_FILE_SERVER= svr_Connection1.trc
```

> **Note:**
>
> Ensure that the name you choose for the `TRACE_FILE_SERVER` file is different from the name you choose for the `TRACE_FILE_CLIENT` file.

E.2.2 Third Party Debugging Tools

You can use tools such as JDBCSpy and JDBCTest from Intersolv to troubleshoot at the JDBC API level. These tools are similar to ODBC Spy and ODBC Test tools.