13
Java Streams in JDBC

This chapter describes how the Oracle Java Database Connectivity (JDBC) drivers handle Java streams for several data types. Data streams enable you to read `LONG` column data of up to 2 gigabytes (GB).

This chapter covers the following topics:

13.1 Overview of Java Streams

Oracle JDBC drivers support the manipulation of data streams in either direction between server and client. The drivers support all stream conversions: binary, ASCII, and Unicode. Following is a brief description of each type of stream:

- Binary

  Used for `RAW` bytes of data, and corresponds to the `getBinaryStream` method
- ASCII

  Used for ASCII bytes in ISO-Latin-1 encoding, and corresponds to the `getAsciiStream` method
- Unicode

  Used for Unicode bytes with the `UTF-16` encoding, and corresponds to the `getUnicodeStream` method

The `getBinaryStream`, `getAsciiStream`, and `getUnicodeStream` methods return the bytes of data in an `InputStream` object.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `CONNECTION_PROPERTY_STREAM_CHUNK_SIZE` is deprecated and the driver does not use it internally for setting the stream chunk size.

> **See Also:**
>
> Working with LOBs and BFILEs

13.2 About Streaming LONG or LONG RAW Columns

This section covers the following topics:

- Overview of Streaming LONG or LONG RAW Columns
- LONG RAW Data Conversions
- LONG Data Conversions
- Examples:Streaming LONG RAW Data
- About Avoiding Streaming for LONG or LONG RAW

13.2.1 Overview of Streaming LONG or LONG RAW Columns

When a query selects one or more `LONG` or `LONG RAW` columns, the JDBC driver transfers these columns to the client in streaming mode. In streaming mode, the JDBC driver does not read the column data from the network for `LONG` or `LONG RAW` columns, until required. The column data remains in the network communications channel until your code calls a `getXXX` method to read the column data. Even after the call, the column data is read only as needed to populate return value from the getXXX call. Because the column data remains in the communications channel, the streaming mode interferes with all other use of the connection. Any use of the connection, other than reading the column data, will discard the column data from the channel. While the streaming mode makes efficient use of memory and minimizes network round trips, it interferes with many other database operations.

> **Note:**
>
> Oracle recommends avoiding `LONG` and `LONG RAW` columns. Use `LOB` instead.

To access the data in a `LONG` column, you can get the column as a Java `InputStream` object and use the `read` method of the `InputStream` object. As an alternative, you can get the data as a `String` or `byte` array. In this case, the driver will do the streaming for you.

You can get `LONG` and `LONG RAW` data with any of the three stream types. The driver performs conversions for you, depending on the character set of the database and the driver.

> **Note:**
>
> Do not create tables with `LONG` columns. Use large object (LOB) columns, `CLOB`, `NCLOB`, and `BLOB`, instead. `LONG` columns are supported only for backward compatibility. Oracle recommends that you convert existing `LONG` columns to `LOB` columns. `LOB` columns are subject to far fewer restrictions than `LONG` columns.

13.2.2 LONG RAW Data Conversions

A call to `getBinaryStream` returns `RAW` data. A call to `getAsciiStream` converts the `RAW` data to hexadecimal and returns the ASCII representation. A call to `getUnicodeStream` converts the `RAW` data to hexadecimal and returns the Unicode characters.

13.2.3 LONG Data Conversions

When you get `LONG` data with `getAsciiStream`, the drivers assume that the underlying data in the database uses an `US7ASCII` or `WE8ISO8859P1` character set. If the assumption is true, then the drivers return bytes corresponding to ASCII characters. If the database is not using an `US7ASCII` or `WE8ISO8859P1` character set, a call to `getAsciiStream` returns meaningless information.

When you get `LONG` data with `getUnicodeStream`, you get a stream of Unicode characters in the `UTF-16` encoding. This applies to all underlying database character sets that Oracle supports.

When you get `LONG` data with `getBinaryStream`, there are two possible cases:

- If the driver is JDBC OCI and the *client* character set is *not* `US7ASCII` or `WE8ISO8859P1`, then a call to `getBinaryStream` returns `UTF-8`. If the *client* character set is `US7ASCII` or `WE8ISO8859P1`, then the call returns a `US7ASCII` stream of bytes.
- If the driver is JDBC Thin and the *database* character set is *not* `US7ASCII` or `WE8ISO8859P1`, then a call to `getBinaryStream` returns `UTF-8`. If the server-side character set is `US7ASCII` or `WE8ISO8859P1`, then the call returns a `US7ASCII` stream of bytes.

> **Note:**
>
> Receiving `LONG` or `LONG RAW` columns as a stream requires you to pay special attention to the order in which you retrieve columns from the database.

The following table summarizes `LONG` and `LONG RAW` data conversions for each stream type.

Table 13-1 LONG and LONG RAW Data Conversions

Data type	BinaryStream	AsciiStream	UnicodeStream
LONG	Bytes representing characters in Unicode `UTF-8`. The bytes can represent characters in `US7ASCII` or `WE8ISO8859P1` if the database character set is `US7ASCII` or `WE8ISO8859P1`.	Bytes representing characters in ISO-Latin-1 (`WE8ISO8859P1`) encoding	Bytes representing characters in Unicode `UTF-16` encoding
LONG RAW	unchanged data	ASCII representation of hexadecimal bytes	Unicode representation of hexadecimal bytes

Related Topics

- Globalization Support

- Relation Between Data Streaming and Multiple Columns

13.2.4 Examples:Streaming LONG RAW Data

One of the features of a `getXXXStream` method is that it enables you to fetch data incrementally. In contrast, `getBytes` fetches all the data in one call. This section contains two examples of getting a stream of binary data. The first version uses the `getBinaryStream` method to obtain `LONG RAW` data, and the second version uses the `getBytes` method.

Getting a LONG RAW Data Column with getBinaryStream

This example writes the contents of a `LONG RAW` column to a file on the local file system. In this case, the driver fetches the data incrementally.

The following code creates the table that stores a column of `LONG RAW` data associated with the name LESLIE:

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

The following Java code snippet writes the data from the `LONG RAW` column into a file called `leslie.gif`:

```
ResultSet rset = stmt.executeQuery
                 ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset.next())
{
    // Get the GIF data as a stream from Oracle to the client
    InputStream gif_data = rset.getBinaryStream (1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie.gif");
      int chunk;
      while ((chunk = gif_data.read()) != -1)
         file.write(chunk);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

In this example, the `InputStream` object returned by the call to `getBinaryStream` reads the data directly from the database connection.

Getting a LONG RAW Data Column with getBytes

This example gets the content of the `GIFDATA` column with `getBytes` instead of `getBinaryStream`. In this case, the driver fetches all the data in one call and stores it in a byte array. The code snippet is as follows:

```
ResultSet rset2 = stmt.executeQuery
                  ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset2.next())
{
   // Get the GIF data as a stream from Oracle to the client
   byte[] bytes = rset2.getBytes(1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie2.gif");
      file.write(bytes);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

Because a `LONG RAW` column can contain up to 2 gigabytes of data, the `getBytes` example can use much more memory than the `getBinaryStream` example. Use streams if you do not know the maximum size of the data in your `LONG` or `LONG RAW` columns.

13.2.5 About Avoiding Streaming for LONG or LONG RAW

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), this method is deprecated.

The JDBC driver automatically streams any `LONG` and `LONG RAW` columns. However, there may be situations where you want to avoid data streaming. For example, if you have a very small `LONG` column, then you may want to avoid returning the data incrementally and, instead, return the data in one call.

To avoid streaming, use the `defineColumnType` method to redefine the type of the `LONG` column. For example, if you redefine the `LONG` or `LONG RAW` column as `VARCHAR` or `VARBINARY` type, then the driver will not automatically stream the data.

If you redefine column types with `defineColumnType`, then you must declare the types of the columns in the query. If you do not declare the types of the columns, then `executeQuery` will fail. In addition, you must cast the `Statement` object to `oracle.jdbc.OracleStatement`.

As an added benefit, using `defineColumnType` saves the OCI driver a database round-trip when running the query. Without `defineColumnType`, these JDBC drivers must request the data types of the column types. The JDBC Thin driver derives no benefit from `defineColumnType`, because it always uses the minimum number of round-trips.

Using the example from the previous section, the `Statement` object `stmt` is cast to `OracleStatement` and the column containing `LONG RAW` data is redefined to be of the type

VARBINARAY. The data is not streamed. Instead, it is returned in a byte array. The code snippet is as follows:

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.OracleStatement ostmt =
   (oracle.jdbc.OracleStatement)stmt;

//redefine the LONG column at index position 1 to VARBINARY
ostmt.defineColumnType(1, Types.VARBINARY);

// Do a query to get the images named 'LESLIE'
ResultSet rset = ostmt.executeQuery
         ("select GIFDATA from streamexample where NAME='LESLIE'");

// The data is not streamed here
rset.next();
byte [] bytes = rset.getBytes(1);
```

13.3 About Streaming CHAR, VARCHAR, or RAW Columns

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), this method is deprecated..

If you use the `defineColumnType` Oracle extension to redefine a `CHAR`, `VARCHAR`, or `RAW` column as a `LONGVARCHAR` or `LONGVARBINARY`, then you can get the column as a stream. The program will behave as if the column were actually of type `LONG` or `LONG RAW`. Note that there is not much point to this, because these columns are usually short.

If you try to get a `CHAR`, `VARCHAR`, or `RAW` column as a data stream without redefining the column type, then the JDBC driver will return a Java `InputStream`, but no real streaming occurs. In the case of these data types, the JDBC driver fully fetches the data into an in-memory buffer during a call to the `executeQuery` method or the `next` method. The `getXXXStream` entry points return a stream that reads data from this buffer.

13.4 About Streaming LOBs and External Files

The term large object (LOB) refers to a data item that is too large to be stored directly in a database table. Instead, a locator is stored in the database table, which points to the location of the actual data. External files are managed similarly. The JDBC drivers can support the following types through the use of streams:

- Binary large object (BLOB)

  For unstructured binary data
- Character large object (CLOB)

  For character data
- National Character large object (NCLOB)

  For national character data
- Binary file (BFILE)

  For external files

LOBs and BFILEs behave differently from the other types of streaming data described in this chapter. Instead of storing the actual data in the table, a locator is stored. The actual data can be manipulated using this locator, including reading and writing the data as a stream. Even when streaming, only the chunk of data (defined by a size) is streamed across the network. By contrast, when streaming a `LONG` or `LONG RAW`, the entire data is streamed across the network.

Streaming BLOBs, CLOBs, and NCLOBs

When a query fetches one or more `BLOB`, `CLOB`, or `NCLOB` columns, the JDBC driver transfers the data to the client. This data can be accessed as a stream. To manipulate `BLOB`, `CLOB`, or `NCLOB` data from JDBC, use methods in the Oracle extension classes `oracle.sql.BLOB`, `oracle.sql.CLOB` and `oracle.sql.NCLOB`. These classes provide specific functionality, such as reading from the `BLOB`, `CLOB`, or `NCLOB` into an input stream, writing from an output stream into a `BLOB`, `CLOB`, or `NCLOB`, determining the length of a `BLOB`, `CLOB`, or `NCLOB`, and closing a `BLOB`, `CLOB`, or `NCLOB`.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the concrete classes in the `oracle.sql` package are deprecated and replaced with the interfaces in the `oracle.jdbc` package. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interfaces.

> **See Also:**
>
> "Data Interface for LOBs"

Streaming BFILEs

An external file, or BFILE, is used to store a locator to a file outside the database. The file can be stored somewhere on the file system of the data server. The locator points to the actual location of the file.

When a query fetches one or more `BFILE` columns, the JDBC driver transfers the file to the client as required. The data can be accessed as a stream To manipulate BFILE data from JDBC, use methods in the Oracle extension class `oracle.sql.BFILE`. This class provides specific functionality, such as reading from the BFILE into an input stream, writing from an output stream into a BFILE, determining the length of a BFILE, and closing a BFILE.

13.5 Relation Between Data Streaming and Multiple Columns

If a query fetches multiple columns and one of the columns contains a data stream, then the contents of the columns following the stream column are not available until the stream has been read, and the stream column is no longer available once any following column is read. Any attempt to read a column beyond a streaming column closes the streaming column.

Streaming Example with Multiple Columns

Consider the following code:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   //get the date data
   java.sql.Date date = rset.getDate(1);

   // get the streaming data
   InputStream is = rset.getAsciiStream(2);

   // Open a file to store the gif data
   FileOutputStream file = new FileOutputStream ("ascii.dat");

   // Loop, reading from the ascii stream and
   // write to the file
   int chunk;
   while ((chunk = is.read ()) != -1)
      file.write(chunk);
   // Close the file
   file.close();

   //get the number column data
   int n = rset.getInt(3);
}
```

The incoming data for each row has the following shape:

```
<a date><the characters of the long column><a number>
```

As you process each row of the result set, you must complete any processing of the stream column before reading the number column.

Bypassing Streaming Data Columns

There may be situations where you want to avoid reading a column that contains streaming data. If you do not want to read such data, then call the `close` method of the stream object. This method discards the stream data and enables the driver to continue reading data from all the columns that contain non-streaming data and follow the column containing streaming data. Even though you are intentionally discarding the stream, it is a good programming practice to retrieve the columns in the same order as in the `SELECT` statement.

In the following example, the stream data in the `LONG` column is discarded and the data from only the `DATE` and `NUMBER` column is recovered:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");

while rset.next()
{
   //get the date
   java.sql.Date date = rset.getDate(1);

   // access the stream data and discard it with close()
   InputStream is = rset.getAsciiStream(2);
   is.close();

   // get the number column data
   int n = rset.getInt(3);
}
```

Related Topics

- About Streaming Data Precautions
- About Streaming LOBs and External Files

13.6 Closing a Stream

You can discard the data from a stream at any time by calling the `close` method. It is a good programming practice to close the stream when you no longer need it. For example:

```
...
InputStream is = rset.getAsciiStream(2);
is.close();
```

> **Note:**
>
> Closing a stream has little performance effect on a LONG or LONG RAW column. All of the data still move across the network and the driver must read the bits from the network.

Related Topics

- Relation Between Data Streaming and Multiple Columns
- About Streaming Data Precautions

13.7 Notes and Precautions on Streams

This section discusses several cautionary issues regarding the use of streams:

- About Streaming Data Precautions
- About Using Streams to Avoid Limits on setBytes and setString
- Relation Between Streaming and Row Prefetching

13.7.1 About Streaming Data Precautions

This section describes some of the precautions you must take to ensure that you do not accidentally discard or lose your stream data. The drivers automatically discard stream data if you perform any JDBC operation that communicates with the database, other than reading the current stream. Two common precautions are:

- Use the stream data after you access it.

  To recover the data from a column containing a data stream, it is not enough to fetch the column. You must immediately process the contents of the column. Otherwise, the contents will be discarded when you fetch the next column.

- Call the stream column in the same order as in the `SELECT` statement.

  If your query fetches multiple columns, the database sends each row as a set of bytes representing the columns in the `SELECT` order. If one of the columns contains stream data, then the database sends the entire data stream before proceeding to the next column.

  If you do not use the order as in the `SELECT` statement to access data, then you can lose the stream data. That is, if you bypass the stream data column and access data in a

column that follows it, then the stream data will be lost. For example, if you try to access the data for the `NUMBER` column *before* reading the data from the stream data column, then the JDBC driver first reads then discards the streaming data automatically. This can be very inefficient if the `LONG` column contains a large amount of data.

If you try to access the `LONG` column later in the program, then the data will not be available and the driver will return a "`Stream Closed`" error.

The later point is illustrated in the following example:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   int n = rset.getInt(3);  // This discards the streaming data
   InputStream is = rset.getAsciiStream(2);
                            // Raises an error: stream closed.
}
```

If you get the stream but do not use it *before* you get the `NUMBER` column, then the stream still closes automatically:

```
ResultSet rset = stmt.executeQuery
                 ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   InputStream is = rset.getAsciiStream(2); // Get the stream
   int n = rset.getInt(3);
   // Discards streaming data and closes the stream
}
int c = is.read(); // c is -1: no more characters to read-stream closed
```

13.7.2 About Using Streams to Avoid Limits on setBytes and setString

Starting from Oracle Database 12*c*, the size limit of the data that is used with the `setBytes` and `setString` methods, have been increased significantly. Any Java `byte` array can be passed to `setBytes`, and any Java `String` can be passed to `setString`. The JDBC driver automatically switches to using `setBinaryStream` or `setCharacterStream` or to using `setBytesForBlob` or `setStringForClob`, depending on the size of the data, whether the statement is SQL or PL/SQL, and the driver used.

There are some limitation with earlier versions of Oracle Database and in the server-side internal driver.

Related Topics

- Data Interface for LOBs
  The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with LOB data types.

13.7.3 Relation Between Streaming and Row Prefetching

If the JDBC driver encounters a column containing a data stream, then row fetch size is set back to one. Row fetch size is an Oracle performance enhancement that enables multiple rows of data to be retrieved with each trip to the database.