16

Working with Large Objects and SecureFiles

Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data. This chapter describes how to use Java Database Connectivity (JDBC) to access and manipulate LOBs and SecureFiles using either the data interface or the locator interface.

> **Note:**
>
> Oracle is deprecating the methods `open()`, `close()`, and `isClosed()` in the interfaces `oracle.jdbc.OracleBlob`, `oracle.jdbc.OracleClob`, and `oracle.jdbc.OracleBfile`. These methods are replaced with the `openLob()`, `closeLob()` and `isClosedLob()` methods. The method `close()` conflicts with the type `java.lang.AutoCloseable`. Removing the proprietary method `close()` makes it possible for `OracleBlob`, `OracleClob`, and `OracleBfile` interfaces to extend the `AutoCloseable` interface at some future time. The `open()` and `isClosed()` methods will be removed and replaced to maintain rational names for these methods.

This chapter contains the following sections:

16.1 The LOB Data Types

Oracle Database supports the following four LOB data types:

- Binary large object (BLOB)

  This data type is used for unstructured binary data.
- Character large object (CLOB)

  This data type is used for character data.
- National character large object (NCLOB)

  This data type is used for national character data.
- BFILE

  This data type is used for large binary data objects stored in operating system files, outside of database tablespaces.

BLOBs, CLOBs, and NCLOBs are stored persistently in a database table and all operations performed on these data types are under transaction control. You can also create temporary LOBs of types BLOB, CLOB, or NCLOB to hold transient data. Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.

BFILE is an Oracle proprietary data type that provides read-only access to data located outside the database tablespaces on tertiary storage devices, such as hard disks, network mounted files systems, CD-ROMs, PhotoCDs, and DVDs. BFILE data is not under transaction control and is not stored by database backups.

The PL/SQL language supports the LOB data types and the JDBC interface allows passing IN parameters to PL/SQL procedures or functions, and retrieval of OUT parameters or returns.

> **See Also:**
>
> Introduction to Large Objects and SecureFiles

16.2 Persistent LOBs

A persistent LOB is a LOB instance that exists in a table row in the database. You can store persistent LOBs as SecureFiles or BasicFiles.

> **See Also:**
>
> Persistent LOBs

SecureFiles is the default storage mechanism for LOBs in database tables. SecureFile LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM). Oracle strongly recommends SecureFiles for storing and managing BLOBs, CLOBs, and NCLOBs.

Following Features of Oracle SecureFiles are transparently available to JDBC programs through the existing APIs:

- Compression enables users to compress data to save disk space.
- Encryption provides an encryption facility that enables random read and write operations of the encrypted data.
- Deduplication enables automatically detect duplicate LOB data and conserve space by storing only one copy of the data.

isSecureFile Method

You can check whether or not your `BLOB` or `CLOB` data uses Oracle SecureFile storage. To achieve this, use the following method from the `oracle.jdbc.OracleBlob` or the `oracle.jdbc.OracleClob` class:

```
public boolean isSecureFile() throws SQLException
```

If this method returns `true`, then your data uses SecureFile storage.

Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.

> **See Also:**
>
> - Data Interface for LOBs
> - Locator Interface for LOBs

16.3 Temporary LOBs

You can use temporary LOBs to store transient data. Temporary LOBs reside in either the PGA memory or the temporary tablespace, depending on their size.

You can insert temporary LOBs into a regular database table. In such a case, a permanent copy of the LOB is created and stored.

> **See Also:**
>
> Temporary LOBs

Creating a Temporary LOB

You create a temporary LOB with the static `createTemporary` method, defined in both the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes. You can also create a temporary LOB by using the connection factory methods available in JDBC 4.0. The Oracle JDBC drivers implement the factory methods, `createBlob`, `createClob`, and `createNClob` in the `java.sql.Connection` interface to create temporary LOBs.

Freeing a Temporary LOB

You free a temporary LOB using the `freeTemporary` method. You can test whether a LOB is temporary or not by calling the `isTemporary` method. If the LOB was created by calling the `createTemporary` method, then the `isTemporary` method returns `true`, else it returns `false`.

Starting with Oracle Database Release 21c, you do not need to check whether a LOB is temporary or persistent before releasing the temporary LOB. If you call the `DBMS_LOB.FREETEMPORARY` procedure or the `OCILobFreeTemporary()` function on a LOB, it will perform either of the following operations:

- For a temporary LOB, it will release the LOB.
- For a persistent LOB, it will do nothing (no-op).

> **Note:**
>
> - You must free any temporary LOBs before ending the session or call. If you do not free a temporary LOB, then it will make the storage used by that LOB in the database unavailable. Frequent failure to free temporary LOBs will result in filling up temporary table space with unavailable LOB storage.
> - The JDBC 4.0 `free` method, present in the `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces, supersedes the `freeTemporary` method.

Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.

> **See Also:**
>
> - Data Interface for LOBs
> - Locator Interface for LOBs

16.4 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.

The data interface uses standard JDBC methods such as the `getString` method and the `setBytes` method to read and write LOB data. Unlike the standard `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces, the data interface does not provide random access capability, that is, it does not use LOB locator and cannot access data beyond a size of 2 gigabytes.

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

You can use the data interface for LOBs to store and manipulate character data and binary data in a LOB column as if it were stored in the corresponding legacy data types like `VARCHAR2`, `LONG`, `RAW`, and so on. This section describes the following topics:

- Input
- Output
- CallableSatement and IN OUT Parameter
- Size Limitations

16.4.1 Input

The `setBytes`, `setBinaryStream`, `setString`, `setCharacterStream`, and `setAsciiStream` methods of the `PreparedStatement` interface are extended to enhance the ability to work with `BLOB`, `CLOB`, and `NCLOB` target columns. If the length of the data is known, then for better performance, use the versions of `setBinaryStream` or `setCharacterStream` methods that accept the data length as a parameter.

> **Note:**
>
> These methods do not work with `BFILE` data because it is read-only.

For the JDBC Oracle Call Interface (OCI) and Thin drivers, there is no limitation on the size of the `byte` array, `String`, or the length specified for the stream functions, except the limits imposed by the Java language.

> **Note:**
>
> In Java, the array size is limited to positive Java `int` or 2 gigabytes of size.

For the server-side internal driver, currently there is a limitation of 32767 bytes for operations on SQL statements, such as an `INSERT` statement. This limitation does not apply for PL/SQL statements. You can use the following workaround for an `INSERT` statement, where you can wrap the LOB in a PL/SQL block:

```
BEGIN
 INSERT id, c INTO clob_tab VALUES(?,?);
END;
```

Input Modes for LOBs

LOBs have the following three input modes:

- Direct binding

  This binding is limited in size but most efficient. It places the data for all input columns inline in the block of data sent to the server. All data, including multiple executions of a batch, is sent in a single network operation.

- Stream binding

  This binding places data at the end. It limits batch size to one and may require multiple round trips to complete.

- LOB binding

  This binding creates a temporary LOB, copies data to the LOB, and binds the LOB locator. The temporary LOB is automatically freed after execution. The operation to create the temporary LOB and then to writing to the LOB requires multiple round trips. The input of the locators may be batched.

You must bear in mind the following automatic switching of the input mode for LOBs:

- For SQL statements:
  - The `setBytes` and `setBinaryStream` methods use direct binding for data less than 32767 bytes.
  - The `setBytes` and `setBinaryStream` methods use stream binding for data larger than 32767 bytes.
  - Starting from JDBC 4.0, there are two forms of `setAsciiStream`, `setBinaryStream`, and `setCharacterStream` methods. The form that accepts a `long` argument as length, uses LOB binding for length larger than 2147483648. The form, where the length is not specified, always uses LOB binding.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use direct binding for data smaller than 32767 characters.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use stream binding for data larger than 32766 characters.
- For PL/SQL statements:

  - The `setBytes` and `setBinary` stream methods use direct binding for data less than 32767 bytes.
  - The `setBytes` and `setBinaryStream` methods use LOB binding for data larger than 32766 bytes.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use direct binding for data smaller than 32767 bytes in the database character set.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use LOB binding for data larger than 32766 bytes in the database character set.

- Forced conversion to LOBs
  The `setBytesForBlob` and `setStringForClob` methods, present in the `oracle.jdbc.OraclePreparedStatement` interface, use LOB binding for any data size.

Impact of Automatic Switching of Input Mode

The automatic switching of the input mode for large data has impact on certain programs. Previously, you used to get `ORA-17157` errors for attempts to use the `setString` method for `String` values larger than 32766 characters. Now, depending on the type of the target parameter, an error may occur while the statement is executed or the operation may succeed.

Another impact is that the automatic switching may result in additional server-side parsing to adapt to the change in the parameter type. This results in a performance effect, if the data sizes vary above and below the limit for repeated executions of the statement. Switching to the stream modes effects batching as well.

16.4.2 Output

The `getBytes`, `getBinaryStream`, `getString`, `getCharacterStream`, and `getAsciiStream` methods of the `ResultSet` and `CallableStatement` interfaces work with `BLOB`, `CLOB`, and `BFILE` columns or `OUT` parameters. These methods work for any `LOB` of length less than 2147483648.

> **Note:**
>
> The `getString` and `getNString` methods cannot be used for retrieving BLOB column values.

The data interface operates by accessing the `LOB` locators within the driver and is transparent to the application programming interface.

You can read `BFILE` data, and read and write `BLOB` or `CLOB` data using the `defineColumnType` method. To read, use the `defineColumnType(nn, Types.LONGVARBINARY)` or the `defineColumnType(nn,Types.LONGVARCHAR)` method on the column. This produces a direct stream on the data as if it were a `LONG RAW` or `LONG` column, and gives the fastest read performance on LOBs.

You can also use LOB prefetching to reduce or eliminate any additional database round trips.

Related Topics

- New Methods for National Character Set Type Data in JDK 6
- Locator Interface for LOBs
  Locators are small data structures, which contain information that may be used to access the actual data of the LOB. In a database table, the locator is stored directly in the table, while the data may be in the table or in separate storage.

16.4.3 CallableSatement and IN OUT Parameter

If you have an `IN OUT CLOB` parameter of a stored procedure and want to use the `setString` method for setting the value for this parameter, then for any `IN` and `OUT` parameter, the binds must be of the same type.

> **Note:**
>
> It is a PL/SQL requirement that the Java types used as input and output for an IN OUT parameter must be the same. The automatic switching of types done by the extensions described in this chapter may cause problems with this.

The automatic switching of the input mode causes problems if you are not sure of the data sizes. For example, if you know that neither the input data nor the output data will ever be larger than 32766 bytes, then you can use the `setString` method for the input parameter and register the `OUT` parameter as `Types.VARCHAR` and use the `getString` method for the output parameter.

A better solution is to change the stored procedure to have separate `IN` and `OUT` parameters. That is, if you have the following piece of code in your application:

```
CREATE PROCEDURE clob_proc( c IN OUT CLOB );
```

then, change it to:

```
CREATE PROCEDURE clob_proc( c_in IN CLOB, c_out OUT CLOB );
```

Another workaround is to use a container block to make the call. The `clob_proc` procedure can be wrapped with a Java String to use for the `prepareCall` statement, as follows:

```
"DECLARE c_temp; BEGIN c_temp := ?; clob_proc( c_temp); ? := c_temp; END;"
```

In either case, you can use the `setString` method on the first parameter and the `registerOutParameter` method with `Types.CLOB` on the second.

16.4.4 Size Limitations

You must be aware of the effect on the performance of the Java memory management system due to creation of a very large `byte` array or a `String`. Read the information provided by your Java virtual machine (JVM) vendor about the impact of very large data elements on memory management, and consider using the stream interfaces instead.

16.5 Locator Interface for LOBs

Locators are small data structures, which contain information that may be used to access the actual data of the LOB. In a database table, the locator is stored directly in the table, while the data may be in the table or in separate storage.

> **See Also:**
>
> Locator Interface for LOBs

Starting from JDBC 4.0, you must use the `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces for performing read and write operations on LOBs. These interfaces provide random access to the data in the LOB.

The Oracle implementation classes `oracle.sql.BLOB`, `oracle.sql.CLOB`, and `oracle.sql.NCLOB` store the locator and access the data with it. The `oracle.sql.BLOB` and `oracle.sql.CLOB` classes implement the `java.sql.Blob` and `java.sql.Clob` interfaces respectively.

> **Note:**
>
> 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 interface.

In Oracle Database 23ai, the Oracle JDBC drivers support the JDBC 4.0 `java.sql.NClob` interface in `ojdbc11.jar`, which is compiled with JDK 11 and the JDBC 4.3 `java.sql.NClob` interface in `ojdbc17.jar`, which is compiled with JDK 17.

In contrast, the `oracle.sql.BFILE` is an Oracle extension, without a corresponding `java.sql` interface.

> **See Also:**
>
> JDBC Javadoc

16.5.1 LOB prefetching

For the current release of JDBC drivers, the number of round trips is reduced by prefetching the metadata such as the LOB length, the chunk size, and the beginning of the LOB data, along with the locator during regular fetch operations.

If you select LOB columns into a result set, then some or all of the data is prefetched to the client, when the locator is fetched. It saves the first round trip to retrieve data by deferring all preceding operations until fetching from the locator.

Note:

- LOBs are not prefetched in abstract data types (ADTs) like STRUCTs and ARRAYs. This behavior is exhibited even if you set the value of the `oracle.jdbc.defaultLobPrefetchSize` connection property. If the client application performs an operation that depends on the value of a LOB, which is embedded in such a data type, then additional round-trips are required to fetch the value.
- The benefits of prefetching are more for small LOBs and less for larger LOBs.
- You must be aware of the possible memory consumption while setting large LOB prefetch sizes in combination with a large row prefetch size and a large number of LOB columns.

The default prefetch size is 32768. You can specify the prefetch size in bytes for BLOBs and in characters for CLOBs, using the `oracle.jdbc.defaultLobPrefetchSize` connection property. You can override the value of this property in the following two ways:

- **At the statement level:** By using the `oracle.jdbc.OracleStatement.setLobPrefetchSize(int)` method
- **At the column level:** By using the form of the `defineColumnType` method that takes length as argument

See Also:

JDBC Javadoc

16.5.2 LOB Open and Close Operations

This section discusses how to open and close your LOBs.

This section discusses how to open and close your LOBs. The JDBC implementation of this functionality is provided using the following methods available in the `oracle.sql.BLOB` and `oracle.sql.CLOB` interfaces:

- `void open (int mode)`
- `void close()`
- `boolean isOpen()`

Note:

You do not have to necessarily open and close your LOBs. You may choose to open and close those for performance reasons.

> **See Also:**
>
> LOB Open and Close Operations

16.6 BFILEs

BFILEs are data objects stored in operating system files, outside the database tablespaces. Data stored in a table column of type BFILE is physically located in an operating system file, not in the database. The BFILE column stores a reference to the operating system file.

BFILEs are read-only. The body of the data resides in the operating system (OS) file system and you can write to BFILEs using only OS tools and commands. You can create a BFILE for an existing external file by executing the appropriate SQL statement using either JDBC APIs or any other way to execute SQL. However, using SQL or JDBC, you cannot create an OS file that a BFILE refers to. Such OS files are created only by an external process that has access to server file systems.

> **See Also:**
>
> BFILEs

This section describes how to use file locators to perform read and write operations on BFILEs. This section covers the following topics:

- Retrieving BFILE Locators
- Inserting BFILES

Retrieving BFILE Locators

Both the BFILE data type and the `oracle.jdbc.OracleBfile` interface to work with the BFILEs are Oracle proprietary. So, there is no standard interface for them. You must use Oracle extensions for this type of data.

If you have a standard JDBC result set or callable statement object that includes BFILE locators, then you can access the locators by using the standard result set `getObject`method. This method returns an `oracle.jdbc.OracleBfile` object.

You can also access the locators by casting your result set to `OracleResultSet` or your callable statement to `OracleCallableStatement` and using the `getOracleObject` or `getBFILE` method.

> **Note:**
>
> If you are using `getObject` or `getOracleObject` methods, then remember to cast the output, as necessary.

Once you have a locator, you can access the BFILE data through the APIs present in the `oracle.jdbc.OracleBfile` class. These APIs are similar to the read methods of the `oracle.jdbc.OracleBfile` interface.

Inserting BFILES

You can use an instance of the `oracle.jdbc.OracleBfile` interface as input to a SQL statement or to a PL/SQL procedure. You can achieve this by performing one of the following:

- Use the standard `setObject` method.
- Cast the statement to `OraclePreparedStatement` or `OracleCallableStatement`, and use the `setOracleObject` or `setBFILE` method. These methods take the parameter index and an `oracle.jdbc.OracleBfile` object as input.

> **Note:**
>
> – There is no standard `java.sql` interface for BFILEs.
>
> – Use the `getBFILE` methods in the `OracleResultSet` and `OracleCallableStatement` interfaces to retrieve an `oracle.jdbc.OracleBfile` object. The `setBFILE` methods in `OraclePreparedStatement` and `OracleCallableStatement` interfaces accept `oracle.jdbc.OracleBfile` object as an argument. Use these methods to write to a BFILE.
>
> – Oracle recommends that you use the `getBFILE`, `setBFILE`, and `updateBFILE` methods instead of the `getBfile`, `setBfile`, and `updateBfile` methods. For example, use the `setBFILE` method instead of the `setBfile` method.

16.7 JDBC Best Practices for LOB

You can enhance the performance of your applications if you reduce the number of round-trips to the database. This section describes how to minimize the number of round-trips to the database.

If you know the maximum size of your LOB data, and you intend to perform read or write operation on the entire LOB, then use the Data Interface following these guidelines:

- For read operations, define the LOB as character type or binary type using the `DefineColumnType` method.
- For write operations, bind the LOB as character type or binary type using the `setString` or the `setBytes` method.

If you do not know the maximum size of your LOB data, then use the LOB APIs with LOB prefetching for read operations. Also, define the LOB prefetch size to a value that can accommodate majority of the LOB values in the column.

> **See Also:**
>
> LOB prefetching