Oracle NoSQL Database

Large Object API

12c Release 1

(Library Version 12.1.3.0)

Legal Notice

7/14/2014

Table of Contents

Using the Large Object API

Oracle NoSQL Database provides an interface you can use to read and write Large Objects (LOBs) such as audio and video files. As a general rule, any object greater than 1 MB is a good candidate for representation as a LOB. The LOB API permits access to large values, without having to materialize the value in its entirety by providing streaming APIs for reading and writing these objects.

A LOB is stored as a sequence of chunks whose sizes are optimized for the underlying storage system. The chunks constituting a LOB may not all be the same size. Individual chunk sizes are chosen automatically by the system based upon its knowledge of the underlying storage architecture and hardware. Splitting a LOB into chunks permits low latency operations across mixed workloads with values of varying sizes. The stream based APIs serve to insulate the application from the actual representation of the LOB in the underlying storage system.

The LOB interface makes use of text-only keys that can be used with either the Tables API or the Key/Value API. This document provides high-level concepts pertinent to the LOB interface, and then provides examples of using it with both the Tables and the Key/Value APIs.

LOB Keys

LOBs are stored and retrieved using `oracle.kv.Key` objects. Each such object contains a series of strings which represent a key path. Key paths are divided into two parts: major and (optionally) minor. The last key path component in a LOB key must end with a trailing suffix that is `.lob` by default.

The LOB suffix may be defined using the `KVStoreConfig.setLOBSuffix()` method.

Unlike other objects stored using `Key` objects (such as when using the Key/Value API), LOB data is not stored on shards driven by the major and minor key path components in the key path. Instead, LOB data uses a hidden keyspace, and its various chunks are distributed across partitions based on this keyspace, instead of based on the `Key` which you provide.

Also, be aware that the actual key you use for your LOBs is stored on a single partition based on its major/minor key path components, and the partition used for this storage is selected in the same way that the store partitions any data based on the key's major and minor key paths. However, assuming reasonably brief keys, this represents a small amount of data and it should not substantially affect your store's data sizing requirements.

The rest of this section provides a brief overview of Oracle NoSQL Database `Keys`, which may be of interest to users of the Table API. Key/Value API users should already be familiar with these concepts and so they can skip to the next section.

A `Key` object can be constructed in several different ways, depending on how you want to represent the key path components. Most commonly, arrays are used to represent the major and minor key path components, and these arrays are then provided to the `Key.createKey()` method. Users of the Table API may find the `Key.fromString()` method convenient because it is easy to store a string representation of a key path in a table cell. Alternatively, Table API users can store key paths components as arrays in table cells, or construct the key path array using information found in table cells.

When represented as a string, key paths begin with a forward slash ('/), which is also used to delimit each path component. If a minor key path is used, then it is delimited from the major key path using a dash ('-). For example, a LOB key used for an image file might be:

```
/Records/People/-/Smith/Sam/Image.lob
```

Users of the Table API should take care to ensure that their LOB paths do not collide with the keys used internally by their tables. In general this is easy to do because the key paths used internally to store table data begin with a numerical representation of the table's name.

LOB Key Checks

All of the LOB APIs verify that the key used to access LOBs meets the trailing suffix requirement. If the trailing suffix requirement is not met, the LOB APIs throw an `IllegalArgumentException` exception. This requirement permits non-LOB methods to check for inadvertent modifications to LOB objects.

This is a summary of LOB-related key checks performed across all methods (LOB and non-LOB):

- When using the Key/Value API, all non-LOB write operations check for the absence of the LOB suffix as part of the other key validity checks. If the check fails (that is, the key contains the LOB suffix), the non-LOB API throws an `IllegalArgumentException`.
- When using the Key/Value API, all non-LOB read operations return the associated opaque value used internally to construct a LOB stream.
- All LOB write and read operations check for the presence of the LOB suffix. If the check fails it will result in an `IllegalArgumentException`.

LOB APIs

Due to their representation as a sequence of chunks, LOBs must be accessed exclusively using the LOB APIs. If you use a LOB key with the family of `KVStore.getXXX` Key/Value methods, you will receive a value that is internal to the KVS implementation and cannot be used directly by the application.

The LOB API is declared in the `KVLargeObject` interface. This is a superinterface of `KVStore`, so you create a `KVStore` handle in the usual way and then use that to access the LOB methods.

The LOB methods are:

- `KVLargeObject.deleteLOB()`

  Deletes a LOB from the store.

- `KVLargeObject.getLOB()`

  Retrieves a LOB from the store.

- `KVLargeObject.putLOB()`, `KVLargeObject.putLOBIfAbsent()`, and `KVLargeObject.putLOBIfPresent()`.

  Writes a LOB to the store.

LOB Operation Exceptions

The methods used to read and insert LOBs are not atomic. This relaxing of the atomicity requirement permits distribution of LOB chunks across the entire store. It is therefore the application's responsibility to coordinate concurrent operations on a LOB. The LOB implementation will make a good faith effort to detect concurrent modification of a LOB and throw `ConcurrentModificationException` when it detects conflicts, but there is no guarantee that the API will detect all such conflicts. The safe course of action upon encountering this exception is to delete the LOB and replace it with a new value after fixing the application level coordination issue that provoked the exception.

Failures during a LOB modification operation result in the creation of a partial LOB. The LOB value of a partial LOB is in some intermediate state, where it cannot be read by the application; attempts to `getLOB()` on a partial LOB will result in a `PartialLOBException`. A partial LOB resulting from an incomplete `putLOB()` operation can be repaired by retrying the operation. Or it can be deleted and a new key/value pair can be created in its place. A partial LOB resulting from an incomplete delete operation must have the delete retried. The documentation associated with individual LOB methods describes their behavior when invoked on partial LOBs in greater detail.

Key/Value LOB Example

The following example writes and then reads a LOB value using the Key/Value API. Notice that the object is never actually materialized within the application; instead, the value is streamed directly from the file system to the store. On reading from the store, this simple example merely counts the number of bytes retrieved.

Also, this example only provides bare-bones exception handling. In production code, you will probably want to do more than simply report the exceptions caught by this example.

```
package kvstore.lobExample;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

import oracle.kv.Consistency;
import oracle.kv.Durability;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.Key;
import oracle.kv.RequestTimeoutException;
import oracle.kv.Version;
import oracle.kv.lob.InputStreamVersion;
import oracle.kv.lob.PartialLOBException;
```

```
public class LOBKV {

    private String[] hhosts = {"localhost:5000"};

    public static void main(String args[]) {
        LOBKV lobkv = new LOBKV();
        lobkv.run(args);
        System.out.println("All done.");
    }
    private void run(String args[]) {

        KVStoreConfig kconfig = new KVStoreConfig("kvstore", hhosts);
        KVStore kvstore = KVStoreFactory.getStore(kconfig);

        // Use key "/test/lob/1.lob" to store the jpeg object.
        // Note that we are not using a minor key in this
        // example. As required, the final key component
        // uses a ".lob" suffix.
        final Key key =
            Key.createKey(Arrays.asList("test", "lob", "1.lob"));

        File lobFile = new File("test.jpg");
        try {
            FileInputStream fis  = new FileInputStream(lobFile);

            // The image file is streamed from the filesystem into
            // the store without materializing it within the
            // application. A medium level of durability is
            // used for this put operation. A timeout value
            // of 5 seconds is set in which each chunk of the LOB
            // must be written, or the operation fails with a
            // RequestTimeoutException.
            kvstore.putLOB(key, fis,
                    Durability.COMMIT_WRITE_NO_SYNC,
                    5, TimeUnit.SECONDS);


            // Now read the LOB. It is streamed from the store,
            // without materialization within the application code.
            // Here, we only count the number of bytes retrieved.
            //
            // We use the least stringent consistency policy
            // available for the read. Each LOB chunk must be read
            // within a 5 second window or a RequestTimeoutException
            // is thrown.
            InputStreamVersion istreamVersion =
                 kvstore.getLOB(key,
                                Consistency.NONE_REQUIRED,
```

```
                                5, TimeUnit.SECONDS);

             InputStream stream = istreamVersion.getInputStream();
             int byteCount = 0;
             while (stream.read() != -1) {
                 byteCount++;
             }
             System.out.println(byteCount);
         } catch (FileNotFoundException fnf) {
             System.err.println("Input file not found.");

             System.err.println("FileNotFoundException: " +
                     fnf.toString());
             fnf.printStackTrace();
             System.exit(-1);
         } catch (RequestTimeoutException rte) {
             System.err.println("A LOB chunk was either not read or");
             System.err.println("not written in the alloted time.");

             System.err.println("RequestTimeoutException: " +
                 rte.toString());
             rte.printStackTrace();
             System.exit(-1);
         } catch (PartialLOBException ple) {
             System.err.println("Retrieval (getLOB()) only retrieved");
             System.err.println("a portion of the requested object.");

             System.err.println("PartialLOBException: " + ple.toString());
             ple.printStackTrace();
             System.exit(-1);
         } catch (IOException e) {
             System.err.println("IO Exception: " + e.toString());
             e.printStackTrace();
             System.exit(-1);
         }
     }

     protected LOBKV() {}
}
```

Table LOB Example

The following example writes and then reads a LOB value using the Table API. Notice that the object is never actually materialized within the application; instead, the value is streamed directly from the file system to the store. On reading from the store, this simple example merely counts the number of bytes retrieved.

When you use LOBs with the Table API, you must still use a `Key` to identify the LOB object. In other words, you cannot directly store the LOB in a table row. Typically you will construct the

Key using information stored in your table. For example, you can simply store the LOB's key as a text string in one of your table cells. Or you could store the key's values as an array in a table cell (or two arrays, if you are using minor key components). Finally, you can construct the key based on values retrieved from one or more cells in the row.

Also, this example only provides bare-bones exception handling. In production code, you will probably want to do more than simply report the exceptions caught by this example.

Before beginning, we must define and create the table in the store. The following table definition describes a very minimal table of user information. It then uses a child table to identify one or more image files associated with the user.

```
table create -name userTable
add-field -type STRING -name userid
add-field -type STRING -name familiarname
add-field -type STRING -name surname
primary-key -field userid -field familiarname -field surname
shard-key -field userid
exit
plan add-table -name userTable -wait

table create -name userTable.images
add-field -type STRING -name imageFileName
add-field -type STRING -name imageVersion
add-field -type STRING -name imageDescription
primary-key -field imageFileName
exit
plan add-table -name userTable.images -wait
```

Add the table definition to the store:

```
> java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar runadmin -host <hostName> \
-port <port> -store <storeName>
kv-> load -file createLOBTable.txt
Table userTable built.
Executed plan 5, waiting for completion...
Plan 5 ended successfully
Table userTable.images built.
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

Now we can write and read table data. In the following example, we create two users that between them have three associated images. First the table rows are created (written), and then the BLOB data is saved to the store. The example then iterates over the tables, showing relevant information, and along the way showing the images associated with each user. In this case, we limit the BLOB display to merely reporting on the BLOB's byte count.

```
package kvstore.lobExample;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

import oracle.kv.Consistency;
import oracle.kv.Durability;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.Key;
import oracle.kv.RequestTimeoutException;
import oracle.kv.Version;
import oracle.kv.lob.InputStreamVersion;
import oracle.kv.lob.PartialLOBException;

import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.MultiRowOptions;

public class LOBTable {

    private String[] hhosts = {"localhost:5000"};

    // Store handles
    private KVStoreConfig kconfig;
    private KVStore kvstore;

    // Table handles
    private TableAPI tableH;
    private Table userTable;
    private Table userImageTable;
    private Row row;

    private static String blobPfx = "blobpfx";
    private static String imgSfx = "userimage.lob";

    public static void main(String args[]) {
        LOBTable lobtable = new LOBTable();
        lobtable.run(args);
        System.out.println("All done.");
    }

    private void run(String args[]) {

        // Open a store handle
```

```
        kconfig = new KVStoreConfig("kvstore", hhosts);
        kvstore = KVStoreFactory.getStore(kconfig);
        tableH = kvstore.getTableAPI();


        // Obtain table handles
        userTable = tableH.getTable("userTable");
        userImageTable = tableH.getTable("userTable.images");

        // populate the tables, and load LOBs into the store
        addData();

        // retrieve tables, and retrieve LOBs from the store
        // and show some details about the tables and LOBs.
        retrieveData();
    }

    // Creates some table rows and loads images into the store
    private void addData() {

        // Load up a couple of rows in the user (parent) table.
        row = userTable.createRow();
        row.put("userid","m.beckstrom.3267");
        row.put("familiarname","Mary");
        row.put("surname","Beckstrom");
        tableH.put(row, null, null);

        row = userTable.createRow();
        row.put("userid","h.zwaska.9140");
        row.put("familiarname","Harry");
        row.put("surname","Zwaska");
        tableH.put(row, null, null);

        // Now populate each row's image (child) table
        // and stream in a BLOB as each row is created.
        row = userImageTable.createRow();
        row.put("userid","m.beckstrom.3267");
        row.put("imageFileName","IMG_2581.jpg");
        row.put("imageDescription","Highrise sunset");
        tableH.put(row, null, null);
        loadBlob("m.beckstrom.3267", "IMG_2581.jpg");

        row = userImageTable.createRow();
        row.put("userid","m.beckstrom.3267");
        row.put("imageFileName","IMG_2607.jpg");
        row.put("imageDescription","Swing set at dawn");
        tableH.put(row, null, null);
        loadBlob("m.beckstrom.3267", "IMG_2607.jpg");
```

```
        row = userImageTable.createRow();
        row.put("userid","h.zwaska.9140");
        row.put("imageFileName","mom1.jpg");
        row.put("imageDescription","Mom's 89th birthday");
        tableH.put(row, null, null);
        loadBlob("h.zwaska.9140", "mom1.jpg");
    }

    // Loads a blob of data into the store
    private void loadBlob(String userid, String filename) {

        // Construct the key.
        // userid and filename are information saved in the
        // table, so later we can recreate the key by table
        // examination. blobPfx is a constant that we use for
        // all BLOB data. imgSfx ends the key path with the
        // required suffix. We use a fixed constant partially
        // to meet the BLOB suffix requirement, but in a
        // larger system this could also be used to
        // differentiate the type of data contained in the
        // BLOB (image data versus an audio file, for example).

        final Key key = Key.createKey(
                Arrays.asList(blobPfx, userid, filename, imgSfx));

        File lobFile = new File(filename);
        try {
            FileInputStream fis  = new FileInputStream(lobFile);
            // The image file is streamed from the filesystem into
            // the store without materializing it within the
            // application. A medium level of durability is
            // used for this put operation. A timeout value
            // of 5 seconds is set in which each chunk of the LOB
            // must be written, or the operation fails with a
            // RequestTimeoutException.
            kvstore.putLOB(key, fis,
                    Durability.COMMIT_WRITE_NO_SYNC,
                    5, TimeUnit.SECONDS);
        } catch (FileNotFoundException fnf) {
            System.err.println("Input file not found.");

            System.err.println("FileNotFoundException: " +
                    fnf.toString());
            fnf.printStackTrace();
            System.exit(-1);
        } catch (RequestTimeoutException rte) {
            System.err.println("A LOB chunk was either not read or");
            System.err.println("not written in the alloted time.");
```

```
            System.err.println("RequestTimeoutException: " +
                rte.toString());
            rte.printStackTrace();
            System.exit(-1);
        } catch (IOException e) {
            System.err.println("IO Exception: " + e.toString());
            e.printStackTrace();
            System.exit(-1);
        }
    }

    // Retrieves the user (parent) table, as well as the images
    // (child) table, and then iterates over the user table,
    // displaying each row as it is retrieved.
    private void retrieveData() {

        PrimaryKey key = userTable.createPrimaryKey();
        // Iterate over every row in the user table including
        // images (child) table in the result set.
        MultiRowOptions mro = new MultiRowOptions(null, null,
                Arrays.asList(userImageTable));
        TableIterator<Row> iter =
                   tableH.tableIterator(key, mro, null);
        try {
            while (iter.hasNext()) {
                Row row = iter.next();
                displayRow(row);
            }
        } finally {
            iter.close();
        }
    }

    // Display a single table row. Tests to see if the
    // table row belongs to a user table or a user images
    // table, and then displays the row appropriately.
    private void displayRow(Row row) {
        if (row.getTable().equals(userTable)) {
            System.out.println("\nName: " +
                    row.get("familiarname").asString().get() +
                    " " +
                    row.get("surname").asString().get());
            System.out.println("UID: " +
                    row.get("userid").asString().get());
        } else if (row.getTable().equals(userImageTable)) {
            showBlob(row);
        }
    }
```

```
// Retrieves and displays a BLOB of data. For this limited
 // example, the BLOB display is limited to simply reporting
 // on its size.
 private void showBlob(Row row) {
     // Build the blob key based on information stored in the
     // row, plus external constants.
     String userid = row.get("userid").asString().get();
     String filename  = row.get("imageFileName").asString().get();
     final Key key = Key.createKey(
             Arrays.asList(blobPfx, userid, filename, imgSfx));

     // Show supporting information about the file which we have
     // stored in the table row:
     System.out.println("\n\tFile: " + filename);
     System.out.println("\tDescription: " +
             row.get("imageDescription").asString().get());

     try {
         // Now read the LOB. It is streamed from the store,
         // without materialization within the application code.
         // Here, we only count the number of bytes retrieved.
         //
         // We use the least stringent consistency policy
         // available for the read. Each LOB chunk must be read
         // within a 5 second window or a RequestTimeoutException
         // is thrown.
         InputStreamVersion istreamVersion =
              kvstore.getLOB(key,
                             Consistency.NONE_REQUIRED,
                             5, TimeUnit.SECONDS);

          InputStream stream = istreamVersion.getInputStream();
          int byteCount = 0;
          while (stream.read() != -1) {
              byteCount++;
          }
          System.out.println("\tBLOB size: " + byteCount);

     } catch (RequestTimeoutException rte) {
         System.err.println("A LOB chunk was either not read or");
         System.err.println("not written in the alloted time.");

         System.err.println("RequestTimeoutException: " +
             rte.toString());
         rte.printStackTrace();
         System.exit(-1);
     } catch (PartialLOBException ple) {
         System.err.println("Retrieval (getLOB()) only retrieved");
         System.err.println("a portion of the requested object.");
```

```
            System.err.println("PartialLOBException: " + ple.toString());
            ple.printStackTrace();
            System.exit(-1);
        } catch (IOException e) {
            System.err.println("IO Exception: " + e.toString());
            e.printStackTrace();
            System.exit(-1);
        }
    }

    protected LOBTable() {}
}
```