Oracle® NoSQL Database

Integrations Guide

Release 20.3
F30919-04
December 2020

Oracle NoSQL Database Integrations Guide, Release 20.3

F30919-04

Contents

9 Appendix

Part III Integration with Oracle Big Data SQL

1 Introduction

Oracle NoSQL Database can be integrated with Apache Hadoop and products in the Oracle stack. The following parts describe more about integration.

Topics

- Part I: Integration with Apache Hadoop MapReduce
- Part II: Integration with Apache Hive
- Part III: Integration with Elastic Search for Full Text Search

Part I

Integration with Apache Hadoop MapReduce

Topics

2

Introduction to Integration with Apache Hadoop MapReduce

This section introduces the integration of Oracle NoSQL Database with Apache Hadoop MapReduce. The information presented in this document describes how MapReduce jobs can be written and run to process data in an Oracle NoSQL Database table. Besides describing the core interfaces and classes involved in this process, this document also walks through an example that demonstrates how to use the Table API, Hadoop integration classes with MapReduce.

The language drivers provide the interfaces and classes that allow MapReduce jobs to be written that retrieve and process table data written to an Oracle NoSQL Database store via the Table API. See Developing for Oracle NoSQL Database in the *Java Direct Driver Developer's Guide*.

Prerequisites

To minimize the number of non-literal text and tokens that need to be replaced when running the examples that are presented, this document assumes that Apache Hadoop and Oracle NoSQL Database are installed on a Big Data Appliance running Big Data SQL 4.0. Specifically, this document assumes that Apache Hadoop is installed under the directory `/opt/cloudera/parcels/CDH`, and that Oracle NoSQL Database is installed under `/opt/oracle/kv-ee`. Thus, if you happen to be using commodity hardware rather than a Big Data Appliance, then you may need to substitute various directory paths presented in this document with values specific to the Apache Hadoop and Oracle NoSQL Database installations on your particular system.

Whether you are using a Big Data Appliance or commodity hardware, in order to work with the examples presented in this document, you will need to install the separate distribution containing the Oracle NoSQL Database Examples. Although you are free to install the example package in any location on your system, for simplicity, this document assumes the example code is installed under the directory `/opt/oracle/nosql/apps/kv/examples` .

> **Note:**
>
> The host names and ports provided below are for demonstration purpose only. You can provide the value as per the requirement.

Before attempting to execute the example that demonstrates the concepts presented in this document, you should first satisfy the following prerequisites:

- Become familiar with Apache Hadoop and the MapReduce programming model. Specifically, become familiar with how to write and deploy a MapReduce job.

- Deploy a Hadoop cluster with 3 DataNodes running on machines with sample host names, `dn-host-1`, `dn-host-2`, and `dn-host-3`.
- Become familiar with Oracle NoSQL Database and then install, start, and configure an Oracle NoSQL Database that is network reachable from the nodes of the Hadoop cluster. The `KVHOME` of the store that you start should be configured as the directory `/opt/oracle/kv-ee`.
- Deploy a store to 3 machines (real or virtual) with sample host names, `kv-host-1`, `kv-host-2`, and `kv-host-3`. The store's name should be set to the value `example-store`, and the store's `KVROOT` should be set to the directories `/u01/nosql/sn1/kvroot` on `kv-host-1`, `/u02/nosql/sn2/kvroot` on `kv-host-2`, and `/u03/nosql/sn3/kvroot` on `kv-host-3`. Finally, an Oracle NoSQL Database admin service, listening on port 5000, should be deployed to each host making up the store.
- Become familiar with the Oracle NoSQL Database Security model and be able to configure the deployed store for secure access (optional). See Introducing Oracle NoSQL Database Security in the *Security Guide*.
- If the deployed store is configured for secure access, start the Oracle NoSQL Database Administrative CLI and securely connect to the store's admin service. See Start the Administration CLI in the *Administrator's Guide*. Using the CLI, create a user in the store named `example-user` along with the appropriate security artifacts (login file, trust file, and either password file or Oracle Wallet [Enterprise Edition only]).
- Obtain and install the separate distribution containing the Oracle NoSQL Database Examples. Although you are free to install that package in any location on your system, for simplicity this document assumes the example code is installed under the directory `/opt/oracle/nosql/apps/kv/examples`.
- Be able to compile and execute a Java program and package it and any associated resources in a JAR file.
- Install the Hadoop JAR files required to compile the example program so that they are available for inclusion in the example program's classpath (see below).

Using specific values for items such as the `KVHOME` and `KVROOT` environment variables, as well as the store name, host names, admin port, and example code location described above should allow you to more easily understand and use the example commands presented in this document. Combined with the information contained in the *Concepts Guide*, along with the *Administrator's Guide* and *Security Guide*, you should then be able to generalize and extend these examples to your particular development scenario; substituting the values specific to the given environment where necessary.

Detailed instructions for deploying a non-secure store are provided in the Deploying a Non-Secure Store appendix of this document. Similarly, the Deploying a Secure Store appendix provides instructions for deploying a store configured for security.

A Brief Primer on Apache Hadoop

Apache Hadoop can be thought of as consisting of two primary components:

- The Hadoop Distributed File System (referred to as, HDFS).
- The MapReduce programming model; which includes a Map Phase consisting of a mapping step and a shuffle-and-sort step that together perform filtering and

sorting, followed by a Reduce Phase that performs a summary operation on the mapped and sorted results from the Map Phase.

The various Hadoop distributions that are available (for example, Cloudera) provide an infrastructure for orchestrating the processing performed in a MapReduce job. This includes marshaling the distributed servers that execute job tasks in parallel, the management of all communication and data transfers between each part of the system, and mechanisms for providing redundancy and fault tolerance.

The Hadoop infrastructure also provides several interactive tools such as a command line interface (the Hadoop CLI) that provide access to the data stored in HDFS. But the typical way application developers read, write, and process data stored in HDFS is via MapReduce jobs; which are programs that adhere to the Hadoop MapReduce programming model. For more detailed information on Hadoop HDFS and MapReduce, see the Hadoop MapReduce tutorial.

As indicated earlier, with the introduction of the Oracle NoSQL Table API, Oracle NoSQL Database provides a set of interfaces and classes that satisfy the Hadoop MapReduce programming model to allow one to write MapReduce jobs that can be run to process data written to a table created in an Oracle NoSQL Database store. These classes are located in the `oracle.kv.hadoop.table` package, and consist of the following types:

- A subclass of the Hadoop class, `org.apache.hadoop.mapreduce.InputFormat`, which specifies how the associated MapReduce job uses a Hadoop `RecordReader` to read its input data and splits the input data into logical sections, each referred to as an `InputSplit`.
- A subclass of the Hadoop class, `org.apache.hadoop.mapreduce.OutputFormat`, which specifies how the associated MapReduce job uses a Hadoop `RecordWriter` to write its output data.
- A subclass of the Hadoop class, `org.apache.hadoop.mapreduce.RecordReader`, which specifies how the mapped keys and values are located and retrieved during MapReduce processing.
- A subclass of the Hadoop class, `org.apache.hadoop.mapreduce.InputSplit`, which represents the data to be processed by an individual MapReduce Mapper; where there is one Mapper per InputSplit.

For the complete list of classes, see Apache Hadoop API.

As described in the following sections, it is through the specific implementation of the `InputFormat` class provided by Oracle NoSQL Database that the Hadoop MapReduce infrastructure obtains access to a given store and the data written to the store.

3
The CountTableRows Example

Assuming you installed the separate example distribution under the directory /opt/oracle/nosql/apps/kv/examples, the hadoop.table example package would contain the following source files under the /opt/oracle/nosql/apps/kv/examples/hadoop/table/ directory:

- CountTableRows.java
- LoadVehicleTable.java
- KVSecurityCreation.java
- KVSecurityUtil.java

To run the MapReduce job launched by the CountTableRows example Java program, an Oracle NoSQL Database store (secure or non-secure) must first be deployed, and a table must be created and populated with data. Thus, before executing CountTableRows, either use the steps outlined in the Deploying a Non-Secure Store appendix to deploy a non-secure store, or use the Deploying a Secure Store appendix to deploy a store configured for security.

Once a store has been deployed, you should execute the standalone Java program LoadVehicleTable provided in the example package to create and populate a table with the name and schema expected by CountTableRows. Once the table is created and populated with example data, CountTableRows can then be executed to run a MapReduce job that counts the number of rows of data in the table.

In addition to the LoadVehicleTable program, the example package also contains the classes KVSecurityCreation and KVSecurityUtil, which are provided to support running CountTableRows against a secure store.

The standalone Java program KVSecurityCreation is provided as a convenience, and can be run to create (or delete) a password file or Oracle Wallet along with associated client side and server side login files that CountTableRows will need to interact with a secure store.

The KVSecurityUtil class provides convenient utility methods that CountTableRows uses to create and process the various security artifacts needed when accessing the store securely.

The CountTableRows Support Programs appendix explains how to compile and execute the LoadVehicleTable program to create and populate the required example table in the store that you deploy. That appendix also explains how to compile and execute the KVSecurityCreation program to create or delete any security credentials that may be needed by CountTableRows.

The following sections explain how to compile, build (JAR), and execute the CountTableRows MapReduce job on the Hadoop cluster that was deployed for this example.

Compile, Build, and Run the `CountTableRows` Example

After you have run the `LoadVehicleTable` program to create and populate the example `vehicleTable` (see the CountTableRows Support Programs appendix), but before you execute the example MapReduce job, you must first compile the `CountTableRows` program and package the compiled artifacts for deployment to the Hadoop infrastructure.

To compile the `CountTableRows` program, several Hadoop JAR files must be installed and available in your build environment for inclusion in the program classpath. Those JAR files are:

- `commons-logging-<version>.jar`
- `hadoop-common-<version>.jar`
- `hadoop-mapreduce-client-core-<version>.jar`
- `hadoop-annotations-<version>.jar`
- `hadoop-yarn-api-<version>.jar`

The `<version>` token used above represents the particular version number of the corresponding JAR file contained in the Hadoop distribution installed in your build environment.

For example, suppose that the `3.0.0` version of Hadoop, packaged by Cloudera version 6.3.0 (`cdh6.3.0`), was installed on your system via parcels; where a parcel is a binary distribution format that Cloudera provides as an alternative to `rpm/deb` packages. Additionally, suppose that the classes from that version of Hadoop use the `1.1.3` version of `commons-logging`. Given these assumptions, to compile the `CountTableRows` program, you would then type the following at the command line:

```
cd /opt/oracle/nosql/apps/kv
javac -classpath \
    /opt/cloudera/parcels/CDH/jars/commons-logging-1.1.3.jar: \
    /opt/cloudera/parcels/CDH/jars/
        hadoop-common-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-mapreduce-client-core-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-annotations-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-yarn-api-3.0.0-cdh6.3.0.jar: \
    /opt/oracle/kv-ee/lib/kvclient.jar:examples \
        examples/hadoop/table/CountTableRows.java
```

This produces the following files:

```
/opt/oracle/nosql/apps/kv/examples/hadoop/table/
    CountTableRows.class
    CountTableRows$Map.class
    CountTableRows$Reduce.class
```

If your specific environment has a different, compatible Hadoop distribution installed, then simply replace the paths and version references in the example command line above with the appropriate values for your particular Hadoop installation.

Building `CountTableRows` When the Store is Non-Secure

If you will be running `CountTableRows` against a non-secure store, then the class files shown in the compilation step presented in the previous section should be placed in a JAR file so that the program can be deployed to the example Hadoop cluster. For example, to create a JAR file containing the class files needed to run `CountTableRows` against data in a non-secure store like that deployed in the Deploying a Non-Secure Store appendix, do the following:

```
cd /opt/oracle/nosql/apps/kv/examples
jar cvf CountTableRows.jar hadoop/table/CountTableRows*.class
```

This produces a JAR file named `CountTableRows.jar`, having the following content, located in the directory `/opt/oracle/nosql/apps/kv/examples`:

```
META-INF/
META-INF/MANIFEST.MF
hadoop/table/CountTableRows.class
hadoop/table/CountTableRows$Map.class
hadoop/table/CountTableRows$Reduce.class
```

Building `CountTableRows` When the Store is Secure

This section explains how to compile all of the Java classes that should be included in the build. If you will be running `CountTableRows` against a secure store like that deployed in the Deploying a Secure Store appendix, in addition to including the `CountTableRows` program, the build also needs to include security credential files as well as the `KVSecurityCreation` program and `KVSecurityUtil` class used to perform various security related functions when executing `CountTableRows`.

To compile the `KVSecurityCreation` and `KVSecurityUtil` classes needed to run the secure version of `CountTableRows`, type the following at the command line:

```
cd /opt/oracle/nosql/apps/kv

javac -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:examples \
    examples/hadoop/table/KVSecurityCreation.java

javac -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:examples \
    examples/hadoop/table/KVSecurityUtil.java
```

Once `KVSecurityCreation` and `KVSecurityUtil` have been compiled, `CountTableRows` itself can be compiled in the same way as that shown in the previous section; that is,

```
javac -classpath \
    /opt/cloudera/parcels/CDH/jars/commons-logging-1.1.3.jar: \
    /opt/cloudera/parcels/CDH/jars/
        hadoop-common-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-mapreduce-client-core-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-annotations-3.0.0-cdh6.3.0.jar: \
    /opt/cloudera/parcels/CDH/jars/ \
        hadoop-yarn-api-3.0.0-cdh6.3.0.jar: \
    /opt/oracle/kv-ee/lib/kvclient.jar:examples \
        examples/hadoop/table/CountTableRows.java
```

The command lines above will produce the following class files:

```
/opt/oracle/nosql/apps/kv/examples/hadoop/table/
    CountTableRows.class
    CountTableRows$Map.class
    CountTableRows$Reduce.class
    KVSecurityUtil.class
    KVSecurityCreation.class
```

Unlike the non-secure case, the build artifacts needed to deploy `CountTableRows` in a secure environment include more than just a single JAR file containing the generated class files. For the secure case, it is necessary to package some artifacts for deployment to the client side of the application that communicates with the store, whereas other artifacts will need to be packaged for deployment to the server side of the application.

Although there are different ways to achieve this "separation of concerns" when deploying a given application, the Model For Building & Packaging Secure Clients appendix of this document presents one particular model you can use to package and deploy the artifacts for applications that will interact with a secure store. With this in mind, the sections in this document related to executing `CountTableRows` against a secure store each assume that the application has been built and packaged according to the instructions presented in the Model For Building & Packaging Secure Clients appendix.

Running `CountTableRows` When the Store is Non-Secure

If you will be running `CountTableRows` against a non-secure store such as that deployed in the Deploying a Non-Secure Store appendix, and you have compiled and built `CountTableRows` in the manner presented in the previous section, the MapReduce job initiated by the `CountTableRows` example program can be deployed by typing the following at the command line of the Hadoop cluster's access node:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/oracle/kv-ee/lib/kvclient.jar
```

```
cd /opt/oracle/nosqlapps/kv

hadoop jar examples/CountTableRows.jar \
    hadoop.table.CountTableRows \
    -libjars \
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/lib/jackson-databind.jar,\
    /opt/oracle/kv-ee/lib/jackson-annotations.jar,\
    example-store \
    kv-host-1:5000 \
    vehicleTable \
    /user/example-user/CountTableRows/vehicleTable/<000N>
```

The Hadoop command interpreter's `-libjars` argument is used to include the third party libraries `kvclient.jar`, `sklogger.jar`, `commonutil.jar`, `failureaccess.jar`, `antlr4-runtime-nosql-shaded.jar`, `jackson-core.jar`, `jackson-databind.jar`, and `jackson-annotations.jar` in the classpath of each MapReduce task executing on the cluster's DataNodes. This is necessary so that those tasks can access classes such as, `TableInputSplit` and `TableRecordReader`, as well as various support classes that are not available on the Hadoop platform.

The value example-store specifies the name of the store you deployed and the value `kv-host-1:5000` specifies the hostname and port to use when connecting to that store. The value vehicleTable specifies the name of the table whose rows will be counted by the MapReduce job. And the last argument, containing the path string, specifies where in the Hadoop HDFS filesystem the final value for the number of rows in the `vehicleTable` should be written.

> **Note:**
>
> The example-user component of the path value input to the last argument corresponds to a directory under the HDFS top-level directory with base path `/user`, which typically corresponds to the user who has initiated the MapReduce job. This directory is usually created in HDFS by the Hadoop cluster administrator. Additionally, the `<000N>` token at the end of the path represents a string such as 0000, 0001, 0002, etc. Although any string can be used for this token, using a different number for "N" on each execution of the job makes it easier to keep track of results when you run the job multiple times.

Running `CountTableRows` When the Store is Secure and a Password File is Used

If you will be running `CountTableRows` against a secure store such as that deployed in the Deploying a Secure Store appendix, and you have compiled, built, and packaged `CountTableRows` and all the necessary artifacts in the manner described in the Model

For Building & Packaging Secure Clients appendix, then `CountTableRows` can be run against the secure store by typing the following at the command line of the Hadoop cluster's access node:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/oracle/kv-ee/lib/kvclient.jar:\
    /opt/oracle/nosql/apps/examples/CountTableRows-pwdServer.jar

cd /opt/oracle/nosqlapps/kv

hadoop jar examples/CountTableRows-pwdClient.jar \
    hadoop.table.CountTableRows \
    -libjars /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/lib/jackson-databind.jar,\
    /opt/oracle/kv-ee/lib/jackson-annotations.jar,\
    /opt/oracle/nosql/apps/examples/CountTableRows-pwdServer.jar \
    example-store \
    kv-host-1:5000 \
    vehicleTable \
    /user/example-user/CountTableRows/vehicleTable/<000N> \
    example-user-client-pwdfile.login \
    example-user-server.login
```

The following items in the command lines above are the client side artifacts of `CountTableRows`,

```
examples/CountTableRows-pwdClient.jar

example-user-client-pwdfile.login
```

whereas the following items are the server side artifacts.

```
/opt/oracle/nosql/apps/examples/CountTableRows-pwdServer.jar

example-user-server.login
```

Rather than using an Oracle Wallet, the mechanism used for storing the user's password is a password file, which is contained in the `CountTableRows-pwdServer.jar` artifact.

Running `CountTableRows` When the Store is Secure and an Oracle Wallet is Used

If you will be running `CountTableRows` against a secure store and you are using an Oracle Wallet rather than a password file to store the user's password, then the

`CountTableRows` MapReduce job can be run by typing the following at the access node's command line:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/oracle/kv-ee/lib/kvclient.jar:\
    /opt/oracle/nosql/apps/examples/CountTableRows-walletServer.jar

cd /opt/oracle/nosqlapps/kv

hadoop jar examples/CountTableRows-walletClient.jar \
    hadoop.table.CountTableRows \
    -libjars \
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar \
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/lib/jackson-databind.jar,\
    /opt/oracle/kv-ee/lib/jackson-annotations.jar,\
    /opt/oracle/nosql/apps/examples/CountTableRows-walletServer.jar \
    example-store \
    kv-host-1:5000 \
    vehicleTable \
    /user/example-user/CountTableRows/vehicleTable/<000N> \
    example-user-client-wallet.login \
    example-user-server.login
```

Whether storing the user's password in a password file or an Oracle Wallet (available in only the Enterprise Edition of Oracle NoSQL Database), notice that an additional JAR file artifact (`CountTableRows-pwdServer.jar` or `CountTableRows-walletServer.jar`) is specified for both the `HADOOP_CLASSPATH` environment variable and the Hadoop `-libjars` parameter. For a detailed explanation of the use and purpose of those server side JAR files, as well as a description of the client side JAR file and the two additional arguments at the end of the command line, refer to the Model For Building & Packaging Secure Clients appendix.

`CountTableRows` MapReduce Job Results

Whether running against a secure or non-secure store, as the job runs, assuming no errors, the output from the job will look like the following:

```
...
INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1344))
    - Running job: job_1409172332346_0024
INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1372))
    -  map 0% reduce 0% INFO  [main] mapreduce.Job
    (Job.java:monitorAndPrintJob(1372))
    -  map 26% reduce 0%
INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1372))
    -  map 56% reduce 0%
INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1372))
    -  map 100% reduce 0% INFO  [main] mapreduce.Job
```

```
    (Job.java:monitorAndPrintJob(1383))
    - Job job_1409172332346_0024 completed successfully
INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1390))
    - Counters: 49
    File System Counters
        FILE: Number of bytes read=2771
        FILE: Number of bytes written=644463
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=2660
        HDFS: Number of bytes written=32
        HDFS: Number of read operations=15
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=6
        Launched reduce tasks=1
        Rack-local map tasks=6
        Total time spent by all maps in occupied slots (ms)=136868
        Total time spent by all reduces in occupied slots (ms)=2103
        Total time spent by all map tasks (ms)=136868
        Total time spent by all reduce tasks (ms)=2103
        Total vcore-seconds taken by all map tasks=136868
        Total vcore-seconds taken by all reduce tasks=2103
        Total megabyte-seconds taken by all map tasks=140152832
        Total megabyte-seconds taken by all reduce tasks=2153472
    Map-Reduce Framework
        Map input records=79
        Map output bytes=2607
        Map output materialized bytes=2801
        Input split bytes=2660
        Combine input records=0
        Combine output records=0
        Reduce input groups=1
        Reduce shuffle bytes=2801
        Reduce input records=79
        Reduce output records=1
        Spilled Records=158
        Shuffled Maps =6
        Failed Shuffles=0
        Merged Map outputs=6
        GC time elapsed (ms)=549
        CPU time spent (ms)=9460
        Physical memory (bytes) snapshot=1888358400
        Virtual memory (bytes) snapshot=6424895488
        Total committed heap usage (bytes)=1409286144
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
```

```
        Bytes Read=0
    File Output Format Counters
        Bytes Written=32
```

To see the results of the job and to verify that the program counted the correct number of rows in the table, use the Hadoop CLI to display the contents of the MapReduce results file located in HDFS. To do this, type the following at the command line of the Hadoop cluster's access node:

```
hadoop fs -cat \
    /user/example-user/CountTableRows/vehicleTable/<000N>/part-r-00000
```

where the `<000N>` token should be replaced with the value you used when the job was run. Assuming the table was populated with 79 rows, if the job was successful, then the output should look like the following:

```
/type/make/model/class  79
```

where `/type/make/model/class` are the names of the fields making up the PrimaryKey of the `vehicleTable`, and 79 is the number of rows in the table.

4
Appendix

Topics

Deploying a Non-Secure Store

The *Concepts Guide*, as well as the *Administrator's Guide*, each presents several different ways to deploy and configure an Oracle NoSQL Database store that does not require secure access. For convenience, this appendix describes one particular set of steps you can take to deploy and configure such a store. Whether you prefer the technique presented here or one of the techniques presented in the concepts manual and administrator's guide, a non-secure store must be deployed and configured to run the example presented in this document in a non-secure environment. For each of the steps presented below, assume the following:

- The Oracle NoSQL Database distribution is installed under the directory `/opt/oracle/kv-ee`.
- A store named `example-store` is deployed to three hosts.
- The hosts are named, `kv-host-1`, `kv-host-2`, and `kv-host-3`, respectively.
- An admin service, listening on port `5000`, is deployed on each of the three hosts.
- The contents of the shards managed by the store will be located under the storage directory `/u01/nosql/sn1/data` for host `kv-host-1`, `/u02/nosql/sn2/data` for host `kv-host-2`, and `/u03/nosql/sn3/data` for host `kv-host-3`.

Given the above assumptions, you can follow the steps below to deploy a non-secure store, where each item from the given assumptions should be replaced with its comparable value specific to your particular environment.

Generate Configuration Files For Each Storage Node (SN)

Login to each host `kv-host-1`, `kv-host-2`, `kv-host-3`, and, from each respective command line, type commands like those shown.

On kv-host-1:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    makebootconfig \
    -root /u01/nosql/sn1/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-1 \
```

```
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u01/nosql/sn1/data \
    -storagedirsize 10000000 \
    -store-security none
```

On kv-host-2:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    makebootconfig \
    -root /u02/nosql/sn2/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-2 \
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u02/nosql/sn2/data \
    -storagedirsize 10000000 \
    -store-security none
```

On kv-host-3:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    makebootconfig \
    -root /u03/nosql/sn3/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-3 \
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u03/nosql/sn3/data \
    -storagedirsize 10000000 \
    -store-security none
```

Launch a Storage Node Agent (SNA) On Each Host Making Up the Store

From each host's command line, type a command like the following:

```
nohup java -jar /opt/oracle/kv-ee/lib/kvstore.jar start \
    -root /u0<n>/nosql/sn<n>/kvroot -config config.xml &
```

where the token <n> corresponds to the integer associated with the given host from which the above command is executed.

Configure and Deploy the Non-secure Store

From the command line of any host that has network connectivity to the nodes making up the store type the following command to enter the store's administrative command line interface (Admin CLI), connected to the boot storage node agent (the boot SNA).

> **Note:**
>
> The node from which you execute the command only requires network connectivity and an Oracle NoSQL Database installation. Thus, although you can execute the command from a separate node that satisfies that requirement, you can also execute that command from any of the nodes making up the store (`kv-host-1`, `kv-host-2`, or `kv-host-3`); as those hosts, by default, satisfy the requirements for launching the Admin CLI.

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar runadmin \
    -helper-hosts kv-host-1:5000,kv-host-2:5000,kv-host-3:5000
```

Once the Admin CLI has been launched, you can deploy the store in one of two ways. First, you can enter the following commands, in succession, at the Admin CLI's command prompt.

```
configure -name example-store
plan deploy-zone -name zn1 -rf 3 -wait

plan deploy-sn -znname zn1 -host kv-host-1 -port 5000 -wait
plan deploy-admin -sn 1 -wait
pool create -name snpool
pool join -name snpool -sn sn1

plan deploy-sn -znname zn1 -host kv-host-2 -port 5000 -wait
plan deploy-admin -sn 2 -wait
pool join -name snpool -sn sn2

plan deploy-sn -znname zn1 -host kv-host-3 -port 5000 -wait
plan deploy-admin -sn 3 -wait
pool join -name snpool -sn sn3

change-policy -params "loggingConfigProps=oracle.kv.level=INFO;"

topology create -name store-layout -pool snpool -partitions 120
topology preview -name store-layout
plan deploy-topology -name store-layout -plan-name deploy-plan -wait
```

Rather than submitting each of the above commands as separate entries to the Admin CLI's command prompt, you may find it more convenient to instead copy each of

those commands to a text file and then specify the single `load` command on the CLI's command prompt; for example,

```
load -file <path-to-command-file>
```

Deploying a Secure Store

The *Security Guide* presents several different ways to deploy and configure an Oracle NoSQL Database store for secure access. For convenience, this section describes one particular set of steps you can take to deploy and configure such a store. Whether you prefer the technique presented here or one of the other techniques presented in the *Security Guide*, a secure store must be deployed and configured in order to run the secure form of the example presented in this document. For each of the steps presented below, in addition to the assumptions made in the non-secure case, also assume the following:

- For convenience, the password manager the store uses to store and retrieve passwords needed for access to keystores and truststores is a password file rather than an Oracle Wallet, which is available in only the Enterprise Edition of Oracle NoSQL Database.
- For simplicity, all passwords are set to the value `No_Sql_00`.
- The name of the user that accesses the store is `example-user`.

Given the above assumptions, you can follow the steps below to deploy a secure store, where each item from the given assumptions should be replaced with its comparable value specific to your particular environment.

Generate Configuration Files For Each Storage Node (SN)

On `kv-host-1`, execute the following command and enter the appropriate responses when prompted:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    makebootconfig \
    -root /u01/nosql/sn1/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-1 \
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u01/nosql/sn1/data \
    -storagedirsize 10000000 \
    -store-security configure \
    -pwdmgr pwdfile \
    -kspwd No_Sql_00

Enter a password for the Java KeyStore: No_Sql_00<RETURN>
Re-enter the KeyStore password for verification: No_Sql_00<RETURN>

Created files
    /u01/nosql/sn1/kvroot/security/store.trust
```

```
    /u01/nosql/sn1/kvroot/security/store.keys
    /u01/nosql/sn1/kvroot/security/store.passwd
    /u01/nosql/sn1/kvroot/security/client.trust
    /u01/nosql/sn1/kvroot/security/security.xml
    /u01/nosql/sn1/kvroot/security/client.security
```

Specifying the value `configure` for the `-store-security` parameter in the above command generates the security artifacts (files) needed for the store's nodes, as well as clients of the store, to communicate securely. Each of the artifacts must be installed on the store's remaining nodes, whereas only the `client.trust` artifact should be installed on any client nodes that will be accessing the store.

To install all of the artifacts listed above on each of the store's remaining nodes, login to each node, create the appropriate `KVROOT` directory, and use a utility such as `scp` to copy the security directory from `kv-host-1` to the given node's `KVROOT` directory. That is,

On kv-host-2:

```
mkdir -p /u02/nosql/sn2/kvroot
cd /u02/nosql/sn2/kvroot
scp -r <username>@kv-host-1:/u01/nosql/sn1/kvroot/security .
```

On kv-host-3:

```
mkdir -p /u03/nosql/sn3/kvroot
cd /u03/nosql/sn3/kvroot
scp -r <username>@kv-host-1:/u01/nosql/sn1/kvroot/security .
```

To install the `client.trust` file on the client node, login to the client node and simply copy the desired file from `kv-host-1` node. That is,

On client-host:

```
scp <username>@kv-host-1:\
    /u01/nosql/sn1/kvroot/security/client.trust /tmp
```

Once the security artifacts generated on `kv-host-1` have been installed on each of the store's remaining nodes, the configuration files for the Storage Nodes that will be deployed to those remaining nodes can be generated. This is accomplished by executing the following commands on the respective node:

On kv-host-2:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    makebootconfig \
    -root /u02/nosql/sn2/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-1 \
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u02/nosql/sn2/data \
    -storagedirsize 10000000 \
    -store-security enable \
```

```
    -pwdmgr pwdfile \
    -kspwd No_Sql_00
```

On kv-host-3:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar makebootconfig \
    -root /u03/nosql/sn3/kvroot \
    -config config.xml \
    -port 5000 \
    -host kv-host-1 \
    -harange 5002,5007 \
    -num_cpus 0 \
    -memory_mb 200 \
    -capacity 1 \
    -storagedir /u03/nosql/sn3/data \
    -storagedirsize 10000000 \
    -store-security enable \
    -pwdmgr pwdfile \
    -kspwd No_Sql_00
```

For both commands above, notice that the value specified for the `-store-security` parameter is `enable` rather than `configure`, which was specified when generating the configuration on `kv-host-1`.

Launch a Storage Node Agent (SNA) On Each Host Making Up the Store

From each host's command line, type a command like the following:

```
nohup java -jar /opt/oracle/kv-ee/lib/kvstore.jar start \
    -root /u0<n>/nosql/sn<n>/kvroot -config config.xml &
```

where the token <n> corresponds to the integer associated with the given host from which the above command is executed.

Configure and Deploy the Secure Store

From the command line of the host `kv-host-1` launch the Admin CLI, connected to the boot SNA.

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar runadmin \
    -helper-hosts kv-host-1:5000,kv-host-2:5000,kv-host-3:5000 \
    -security /u01/nosql/sn1/kvroot/security/client.security
```

Next, from the Admin CLI's command prompt, deploy the store by either entering each command shown below in succession or by using the `load -file <file>` command to load those same commands from a file.

```
configure -name example-store
plan deploy-zone -name zn1 -rf 3 -wait

plan deploy-sn -znname zn1 -host kv-host-1 -port 5000 -wait
```

```
plan deploy-admin -sn 1 -wait
pool create -name snpool
pool join -name snpool -sn sn1

plan deploy-sn -znname zn1 -host kv-host-2 -port 5000 -wait
plan deploy-admin -sn 2 -wait
pool join -name snpool -sn sn2

plan deploy-sn -znname zn1 -host kv-host-3 -port 5000 -wait
plan deploy-admin -sn 3 -wait
pool join -name snpool -sn sn3

change-policy -params "loggingConfigProps=oracle.kv.level=INFO;"

topology create -name store-layout -pool snpool -partitions 120
topology preview -name store-layout
plan deploy-topology -name store-layout -plan-name deploy-plan -wait

execute "CREATE USER root IDENTIFIED BY 'No_Sql_00' ADMIN";
```

> **Note:**
>
> The only difference between the set of store deployment commands presented in the Configure and Deploy the Non-secure Store appendix for a non-secure store and the commands above is the last command. Once the store is deployed, that last command will create a store user named `root` with administrative privileges and password equal to the value `No_Sql_00`.

When a secure store is deployed, before the store can be used, an initial user must be created and then provisioned with the necessary security credentials that grant that user privileges that allow it to administer the store. Once that user is created and provisioned, it can then be used to create other users of the store. In a typical production scenario, tables are generally created and populated with data by users with only user-level privileges rather than administrative privileges.

The last command above then simply creates that initial user that will be used to create a second user for executing the secure version of the example program presented in this document. But before that root user can create other users of the store, it must first be provisioned, as explained in the next section.

Provision the Secure Store's Administrative User (root)

As described in the previous section, the last step of the secure store deployment process simply creates the store's administrative user but does not provision it. But in order to administer the store, that new user must be provisioned with credentials that grant administrative privileges for the store. To provision the root user created in the previous section, login to the store's `kv-host-1` node, execute the commands shown, and enter the appropriate responses when prompted:

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    securityconfig pwdfile create \
```

```
    -file /u01/nosql/sn1/kvroot/security/root.passwd

Created

java -jar /opt/oracle/kv-ee/lib/kvstore.jar \
    securityconfig pwdfile secret \
    -file /u01/nosql/sn1/kvroot/security/root.passwd \
    -set -alias root

Enter the secret value to store: No_Sql_00<RETURN>
Re-enter the secret value for verification: No_Sql_00<RETURN>

Secret created
OK

cp /u01/nosql/sn1/kvroot/security/client.security \
    /u01/nosql/sn1/kvroot/security/root.login

echo oracle.kv.auth.username=root >> \
    /u01/nosql/sn1/kvroot/security/root.login
echo oracle.kv.auth.pwdfile.file=\
    /u01/nosql/sn1/kvroot/security/root.passwd >> \
    /u01/nosql/sn1/kvroot/security/root.login
```

The `client.security` properties file is one of the security artifacts that was generated in the Generate Configuration Files For Each Storage Node (SN) appendix. The contents of that file are copied to the file named `root.login`. The `root.login` file created here is used when clients wishing to connect to the secure store must authenticate as the user named `root`. For the purposes of this document, this authentication process will be referred to as logging in to the secure store. As a result, the properties file used in that authentication process is referred to as a login file, or login properties file.

For convenience, the system properties `oracle.kv.auth.username` and `oracle.kv.auth.pwdfile.file` are inserted into the `root.login` file. This will allow the client to connect to the secure store as the root user without having to specify the value of those properties on the command line.

Create Non-Administrative User

To create a user that will be provisioned with non-administrative privileges, from the store's `kv-host-1` node, login to the Admin CLI as the newly created root user.

```
java -jar /opt/oracle/kv-ee/lib/kvstore.jar runadmin \
    -host kv-host-1 \
    -port 5000 \
    -security /u01/nosql/sn1/kvroot/security/root.login
```

Then create a custom role with the name `readwritemodifytables` (for example) that consists of the privileges a user would need to create and populate a table in the store. After creating the desired role, create a user named `example-user` and grant the `readwritemodifytables` role to that user. To accomplish this, either enter each command shown below in succession or copy each command to a text file

and execute the CLI's load command, specifying the file you created ('`load -file <file>`').

```
execute 'CREATE ROLE readwritemodifytables'
execute 'GRANT SYSDBA TO readwritemodifytables'
execute 'GRANT READ_ANY TO readwritemodifytables'
execute 'GRANT WRITE_ANY TO readwritemodifytables'
execute 'CREATE USER example-user IDENTIFIED BY "No_Sql_00"'
execute 'GRANT readwritemodifytables TO USER example-user'
```

> **Note:**
>
> The name of the user created above is not required to be the same as the OS user name under which the example is executed. The name above and its associated credentials are registered with the secure store for the purpose of authenticating to the store. Thus, the name of the user that is created here can be any value you wish to use.

Provision the Secure Store's Non-Administrative User (example-user)

Once the user named `example-user` and its role have been created, use the `KVSecurityCreation` convenience program to generate the public and private credentials needed by that user to connect to the secure store. To do this, first compile `KVSecurityCreation` by executing the following command from the store's `kv-host-1` node:

```
cd /opt/oracle/nosql/apps/kv
javac -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:examples \
    examples/hadoop/table/KVSecurityCreation.java
```

This will produce the following class files on the `kv-host-1` node:

```
/opt/oracle/nosql/apps/kv/examples/hadoop/table/
    KVSecurityUtil.class
    KVSecurityCreation.class
```

Once `KVSecurityCreation` has been compiled, it can then be executed to generate the desired security artifacts for the non-administrative user. If you want to store the password in a clear text password file, then type the following at the command line and enter the appropriate response when prompted:

```
cd /opt/oracle/nosql/apps/kv
java -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.table.KVSecurityCreation \
    -pwdfile example-user.passwd \
    -set -alias example-user
```

```
INFO: removed file [/tmp/example-user.passwd]
INFO: removed file [/tmp/example-user-client-pwdfile.login]
created login properties file [/tmp/example-user-client-pwdfile.login]
created login properties file [/tmp/example-user-server.login]
created credentials store [/tmp/example-user.passwd]

Enter the secret value to store: No_Sql_00<RETURN>
Re-enter the secret value for verification: No_Sql_00<RETURN>

Secret created
OK
```

Alternatively, if you are using an Oracle Wallet (available only in the Enterprise Edition) to store the user's password, then type the following and again, enter the appropriate response when prompted:

```
cd /opt/oracle/nosql/apps/kv
java -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.table.KVSecurityCreation \
    -wallet example-user-wallet.dir \
    -set -alias example-user

INFO: removed file [/tmp/example-user-wallet.dir/cwallet.sso]
INFO: removed directory [/tmp/example-user-wallet.dir]
INFO: removed file [/tmp/example-user-client-wallet.login]
created login properties file [/tmp/example-user-client-wallet.login]
created login properties file [/tmp/example-user-server.login]
created credentials store [/tmp/example-user-wallet.dir]

Enter the secret value to store: No_Sql_00<RETURN>
Re-enter the secret value for verification: No_Sql_00<RETURN>

Secret created
OK
```

Compare the artifacts generated when a password file is specified with the artifacts generated when a wallet is specified. When a password file is specified, you should see the following files:

```
/tmp
    example-user-client-pwdfile.login
    example-user-server.login
    example-user.passwd
```

Whereas when wallet storage is specified, you should see:

```
/tmp
    example-user-client-wallet.login
    example-user-server.login
```

```
/example-user-wallet.dir
    cwallet.sso
```

> **Note:**
>
> As this is an example for demonstration purposes, the credential files generated by `KVSecurityCreation` are placed in the system's `/tmp` directory. For your applications, you may want to place the credential files you generate in a more permanent location that is password protected.

> **Note:**
>
> For both the password or wallet cases two login properties files are generated; one for client side connections, and one for server side connections. The only difference between the client side login file and the server side login file is that the client side login file specifies the username (the alias) along with the location of the user's password. For the login properties file associated with the use of a password file, the property `oracle.kv.auth.pwdfile` is used to specify the location of the file in which the user's password is stored; whereas the property `oracle.kv.auth.wallet.dir` would be used if the password is stored in an Oracle Wallet. Although optional, the reason for using two login files is to avoid passing private security information to the server side, as explained in more detail in the Model For Building & Packaging Secure Clients appendix. Additionally, observe that the server side login file (`example-user-server.login`) is identical for both cases. This is because whether a password file or an Oracle Wallet is used to store the password, both use the same publicly visible communication transport information.

At this point, the store has been deployed, configured for secure access, and provisioned with the necessary users and credentials required for table creation and population. To demonstrate running a MapReduce job against table data contained in a secure store, the example presented in this document can now be executed by a user whose password is stored either in a clear text password file or an Oracle Wallet (Enterprise Edition only).

> **Note:**
>
> A final, important point is that the storage mechanism used for the example application's user password (password file or Oracle Wallet) does not depend on the password storage mechanism used by the store. That is, although this appendix (for convenience) deployed a secure store using a password file rather than a wallet, the fact that the store placed the passwords it manages in a password file does not prevent the developer/deployer of a client of that store from storing the client's user password in an Oracle Wallet, or vice versa. You should therefore view the use of an Oracle Wallet or a password file by any client application as simply a "safe" place (for some value of "safe") where the user password can be stored and accessed by only the user who owns the wallet or password file. This means that the choice of password storage mechanism is at the discretion of the application developer/deployer, no matter what mechanism is used by the store itself.

`CountTableRows` Support Programs

Oracle NoSQL Database provides a separate distribution in Oracle Technology Network consisting of example programs and utility classes that you can use to explore various aspects of interacting with an Oracle NoSQL Database system. With respect to exploring the integration of Oracle NoSQL Database with MapReduce, in addition to providing the `CountTableRows` example program presented in this document, the Oracle NoSQL Database examples also provide the `LoadTableVehicle` program that you can use to create and populate an example table in the store you deploy.

The sections below describe the `LoadVehicleTable` program; including the schema employed when creating the table, as well as how to compile and execute the program.

Schema for the `vehicleTable` Example

To execute the `CountTableRows` MapReduce job, a table named `vehicleTable` having the schema shown in the table below must be created in the Oracle NoSQL Database store deployed for this example. The data types specified in the schema shown below are defined by the Oracle NoSQL Database Table API (see `oracle.kv.table.FieldDef.Type`) .

Table 4-1 Schema for vehicleTable

Field Name	Field Type	Primary Key	Shard Key
type	FieldDef.Type.STRING	Y	Y
make	FieldDef.Type.STRING	Y	Y
model	FieldDef.Type.STRING	Y	Y
class	FieldDef.Type.STRING	Y	
color	FieldDef.Type.STRING		
price	FieldDef.Type.DOUBLE		

Table 4-1 (Cont.) Schema for vehicleTable

Field Name	Field Type	Primary Key	Shard Key
count	FieldDef.Type.INTEGER		
dealerid	FieldDef.Type.NUMBER		
delivered	FieldDef.Type.TIMESTAMP		

The example `vehicleTable` consists of rows representing a particular vehicle a dealer might have in stock for purchase. Each such row contains fields specifying the "type" of vehicle (for example, car, truck, SUV, etc.), the "make" of the vehicle (Ford, GM, Chrysler, etc.), the "model" (Explorer, Camaro, Lebaron, etc.), the vehicle "class" (4WheelDrive, FrontWheelDrive, etc.), the "color" and "price" of the vehicle, the number of vehicles currently in stock (the "count") having those characteristics, a number that uniquely identifies the dealership selling those vehicles (the "dealerid"), and finally, the date and time those vehicles were "delivered" to the dealership.

Although you can enter individual commands in the store's admin CLI to create a table with the above schema, the preferred approach is to employ the Table Data Definition Language (DDL) to create the desired table. One way to accomplish this is to follow the instructions presented in the next sections to compile and execute the `LoadVehicleTable` program, which will populate the desired table after using the DDL to create it.

Create and Populate `vehicleTable` with Example Data

Assuming an Oracle NoSQL Database store (secure or non-secure) has been deployed with `KVHOME` equal to `/opt/oracle/kv-ee`, the `LoadVehicleTable` program that is supplied as a convenience with the `CountTableRows` example can be executed to create and populate the table named `vehicleTable`. Before executing `LoadVehicleTable` though, that program must first be compiled. To do this, assuming you have installed the example distribution under the base directory `/opt/oracle/nosql/apps/kv/examples`, type the following from your client node's OS command line:

```
cd /opt/oracle/nosql/apps/kv
javac -classpath \
    /opt/oracle/kv-ee/lib/kvclient.jar:examples \
    examples/hadoop/table/LoadVehicleTable.java
```

This should produce the file:

```
/opt/oracle/nosql/apps/kv/examples/hadoop/table/LoadVehicleTable.class
```

Run `LoadVehicleTable` when the Store is Non-Secure

To execute `LoadVehicleTable` to create and populate the table named `vehicleTable` with example data in a store configured for non-secure access, type the following at the command line of the client node, which must have network connectivity with a

node running the admin service of the non-secure store you deployed (for example, `kv-host-1` itself):

```
cd /opt/oracle/nosql/apps/kv
java -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.table.LoadVehicleTable -store example-store \
    -host kv-host-1 -port 5000 -nops 79 [-delete]
```

The following parameters are required: `-store`, `-host`, `-port`, and `-nops`, whereas the `-delete` parameter is optional.

In the example command line above, the argument `-nops 79` requests that 79 rows be written to the `vehicleTable`. If more or less than that number of rows is desired, then the value of the `-nops` parameter should be changed.

If `LoadVehicleTable` is executed a second time and the optional `-delete` parameter is specified, then all rows added by any previous executions of `LoadVehicleTable` are deleted from the table prior to adding the requested new rows. Otherwise, all pre-existing rows are left in place, and the number of rows in the table will be increased by the requested `-nops` number of new rows.

> **Note:**
>
> Because of the way `LoadVehicleTable` generates records, it is possible that a given record has already been added to the table, either during a previous call to `LoadVehicleTable`, or during the current call. As a result, it is not uncommon for the number of unique rows added to be less than the number requested. Because of this, when processing has completed, `LoadVehicleTable` will display the number of unique rows that are actually added to the table, along with the total number of rows currently in the table (from previous runs).

Run `LoadVehicleTable` When the Store is Secure

To execute `LoadVehicleTable` against the secure store that you deployed and provisioned with a non-administrative user according to the steps presented in the Deploying a Secure Store appendix, an additional parameter must be added to the command line above. In this case, type the following on the command line:

```
scp <username>@kv-host-<n>:\
    /u01/nosql/sn1/kvroot/security/client.trust /tmp

cd /opt/oracle/nosql/apps/kv

java -classpath \
    /opt/oracle/kv-ee/lib/kvclient.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.table.LoadVehicleTable -store example-store \
```

```
    -host kv-host-1 -port 5000 -nops 79 \
    -security /tmp/example-user-client-pwdfile.login \
    [-delete]
```

The `client.trust` file generated when the secure store was deployed must be installed in the `/tmp` directory of the client node from which `LoadVehicleTable` is executed. If the client node is different than any of the store nodes (`kv-host-1`, `kv-host-2`, `kv-host-3`), then the installation of `client.trust` is accomplished by performing a remote copy; using the appropriate username and the number 1 in place of the <n> token. On the other hand, if `LoadVehicleTable` is run from one of the nodes making up the store itself, then a local copy operation can be used for the installation.

The additional `-security` parameter in the command above specifies the location of the login properties file (associated with a password file in this case rather than an Oracle Wallet) for the given user or alias. All other parameters are the same as for the non-secure case.

To understand the `-security` parameter, recall from the Deploying a Secure Store appendix that a non-administrative user named `example-user` was created, and a number of credential files based on a password file (rather than an Oracle Wallet) were generated for that user and placed under the `/tmp` system directory. As a result, you should see the following files under the `/tmp` directory of the client node:

```
/tmp
    client.trust
    example-user-client-pwdfile.login
    example-user-server.login
    example-user.passwd
```

For this example, the user credential files must be co-located, where it doesn't matter which directory they are located in, as long as they all reside in the same directory accessible by the user. It is for this reason that the shared trust file (`client.trust`) is copied into `/tmp` above. Co-locating `client.trust` and `example-user.passwd` with the login file (`example-user-client-pwdfile.login`) allows relative paths to be used for the values of the system properties `oracle.kv.ssl.trustStore` and `oracle.kv.auth.pwdfile.file` that are specified in the login file (or `oracle.kv.auth.wallet.dir` if an Oracle Wallet is used to store the user password). If those files are not co-located with the login file, then absolute paths must be used for those properties.

Summary

At this point, the `vehicleTable` created in the Oracle NoSQL Database store you deployed whether non-secure or secure should be populated with the desired example data. And the MapReduce job initiated by `CountTableRows` can be run to count the number of rows in that table.

Model For Building & Packaging Secure Clients

With respect to running a MapReduce job against data contained in a secure store, a particularly important issue to address is related to the communication of user credentials to the tasks run on each of the DataNodes on which the Hadoop infrastructure executes the job. Recall from above that when using the MapReduce

programming model defined by Apache Hadoop the tasks executed by a MapReduce job each act as a client of the store. Thus, if the store is configured for secure access, in order to retrieve the desired data from the store, each task must have access to the credentials of the user associated with that data. The typical mechanism for providing the necessary credentials to a client of a secure store is to manually install the credentials on the client's local file system; for example, by employing a utility such as `scp`.

Although the manual mechanism is practical for most clients of a secure store, it is extremely impractical for a MapReduce job. This is because a MapReduce job consists of multiple tasks running in parallel, in separate address spaces, each with a separate file system that is generally not under the control of the user. Assuming then, that write access is granted by the Hadoop administrator (a problem in and of itself), this means that manual installation of the client credentials for every possible user known to the given secure store would need to occur on the file system of each of the many nodes in the Hadoop cluster; something that may be very difficult to achieve.

To address this issue, a model will be presented that developers and deployers can employ to facilitate the communication of each user's credentials to a given MapReduce job from the client side of the job; that is, from the address space controlled by the job's client process, owned by the user.

This model will consist of two primary components: a programming model for executing MapReduce jobs that retrieve and process data contained in tables located in a secure store; and a set of "best practices" for building, packaging, and deploying those jobs. Although there is nothing preventing a user from manually installing the necessary security credentials on all nodes in a given cluster, doing so is not only impractical, but may result in various security vulnerabilities. Combining this programming model with the deployment best practices that are presented here should help developers and deployers not only avoid the need to manually pre-install credentials on the DataNodes of the Hadoop cluster, but should also prevent the sort of security vulnerabilities that can occur with manual installation.

Programming Model For MapReduce with Oracle NoSQL Database Security

Recall that when executing a MapReduce job, the client application uses mechanisms provided by the Hadoop infrastructure to initiate the job from a node (referred to as the Hadoop cluster's access node) that has network access to the node running the Hadoop cluster's ResourceManager. If the job will be run against a secure store, then prior to initiating the job, the client must initialize the job's `TableInputFormat` with the following three pieces of information:

- The name of the file that specifies the transport properties the client will use when connecting to the store; which, for the purposes of this document, will be referred to as the login properties file (or login file).
- The PasswordCredentials containing the username and password the client will present to the store during authentication.
- The name of the file containing the public keys and/or certificates needed for authentication; which, for the purposes of this document, will be referred to as, the client trust file (or trust file).

To perform this initialization of the MapReduce client application, `CountTableRows` in this case, invokes the `setKVSecurity` method defined in `TableInputFormat`. Once

this initialization has been performed and the job has been initiated, the job uses that `TableInputFormat` to create and assign a `TableInputSplit` (a split) to each of the Mapper tasks that will run on one of the DataNodes in the cluster. The `TableInputFormat` needs the information initialized by the `setKVSecurity` method for two reasons:

- To connect to the secure store from the access node and retrieve the information needed to create the splits.
- To initialize each split with that same security information, so that each such split can connect to the secure store from its DataNode host and retrieve the particular table data the split will process.

In addition to requiring that the MapReduce application use the mechanism just described to initialize and configure the job's `TableInputFormat` (and thus, its splits) with the information listed above, the model also requires that the public and private security credentials referenced by that information be communicated to the `TableInputFormat`, as well as the splits, securely. How this is achieved depends on whether that information is being communicated to the `TableInputFormat` on the client side of the application, or to the splits on the server side.

Communicating Security Credentials to the Server Side Splits

To facilitate communication of the user's security credentials to the splits distributed to each of the DataNodes of the cluster, the model presented here separates public security information from the private information (the username and password), and then stores the private information as part of each split's internal state, rather than on the local file system of each associated DataNode; which may be vulnerable or difficult/impossible to secure. For communication of the public contents of the login and trust files to each such split, the model supports an (optional) mechanism that allows the application to communicate that information as Java resources that each split retrieves from the classpath of the split's Java VM. This avoids the need to manually transfer the contents of those files to each DataNode's local file system, and also avoids the potential security vulnerabilities that can result from manual installation on those nodes. Note that when an application wishes to employ this mechanism, it will typically include the necessary information in a JAR file that is specified to the MapReduce job via the Hadoop command line directive `-libjars`.

The intent of the mechanism just described is to allow applications to exploit the Hadoop infrastructure to automatically distribute the public login and trust information to each split belonging to the job via a JAR file added to the classpath on each remote DataNode. But it is important to note that although this mechanism is used to distribute the application's public credentials, it must not be used to distribute any of the private information related to authentication; specifically, the username and password. This is important because a JAR file that is distributed to the DataNodes in the manner described may be cached on the associated DataNode's local file system; which might expose a vulnerability. As a result, private authentication information is only communicated as part of each split's internal state.

The separation of public and private credentials supported by this model not only prevents caching the private credentials on each DataNode, but also facilitates the ability to guarantee the confidentiality of that information, via whatever external third party secure communication mechanism the current Hadoop implementation happens to employ. This capability is also important to support the execution of Hive queries against a secure store.

Communicating Security Credentials to the `TableInputFormat`

With respect to the job's `TableInputFormat`, the programming model supports different options for communicating the user's security information. This is because the `TableInputFormat` operates only on the access node, on the client side of the job; which means that there is only one file system that needs to be secured. Additionally, unlike the splits, the `TableInputFormat` is not sent on the wire. Thus, as long as only the user is granted read privileges, both the public and private security information can be installed on the access node's file system without fear of compromise. For this case, the application would typically use system properties on the command line to specify the fully-qualified paths to the login, trust, and password files (or Oracle Wallet); which the `TableInputFormat` would then read from the local file system, retrieving the necessary public and private security information.

A second option for communicating the user's security credentials to the `TableInputFormat` is to include the public and private information as resources in the client side classpath of the Java VM in which the `TableInputFormat` runs. This is the option employed by the example presented in this document, and is similar to what was described above for the splits. This option demonstrates how an application's build model can be exploited to simplify not only the applications's command line, but also the deployment of secure MapReduce jobs in general. As was the case with the splits, applications will typically communicate the necessary security information as Java resources by including that information in a JAR file. But rather than using the Hadoop command line directive `-libjars` to specify the JAR file to the server side of the MapReduce job, in this case, because the `TableInputFormat` operates on only the client side access node, the JAR file would simply be added to the `HADOOP_CLASSPATH` environment variable.

Best Practices: MapReduce Application Packaging for Oracle NoSQL Security

To help users achieve the sort of separation of public and private security information described in previous sections, a set of (optional) best practices related to packaging the client application and its necessary artifacts is presented in this section, and are employed by the example featured in this document. Although the use of these packaging practices is optional, you are encouraged to employ them when working with any MapReduce jobs of your own that will interact with a secure store.

Rather than manually installing the necessary security artifacts (login file, trust file, password file or Oracle Wallet) on each DataNode in the cluster, user's should instead install those artifacts only on the cluster's single access node; the node from which the client application is executed. The client application can then retrieve each artifact from the local environment, repackage the necessary information, and then employ mechanisms provided by the Hadoop infrastructure to transfer that information to the appropriate components of the MapReduce job that will be executed.

For example, as described in the previous section, your client application can be designed to retrieve the username and location of the password from the command line, a configuration file, or a resource in the client classpath; where the location of the user's password is a locally installed password file or Oracle Wallet that can only be read by the user. After retrieving the username from the command line and the password from the specified location, the client uses that information to create the user's PasswordCredentials, which are transferred to each MapReduce task via the

splits that are created by the job's `TableInputFormat`. Using this model, the user's PasswordCredentials, are never written to the file systems of the cluster's DataNodes. They are only held in each task's memory. As a result, the integrity and confidentiality of those credentials only need to be provided when on the wire, which can be achieved by using whatever external third party secure communication mechanism the current Hadoop implementation happens to employ.

With respect to the transfer of the public login and trust artifacts, the client application can exploit the mechanisms provided by the Hadoop infrastructure to automatically transfer classpath (JAR) artifacts to the job's tasks. As demonstrated by the `CountTableRows` example presented in the body of this document, the client application's build process can be designed to separate the application's class files from its public security artifacts. Specifically, the application's class files and optionally, the public and private credentials, can be placed in a local JAR file on the access node for inclusion in the classpath of the client itself; while only the public login properties and client trust information are placed in a separate JAR file that can be added to the hadoop command line specification of `-libjars` for inclusion in the classpath of each MapReduce task.

Application Packaging for the Non-Secure Case

To understand how the packaging model discussed here can be employed when executing an application against a secure store, it may be helpful to first review how the `CountTableRows` example is executed against a non-secure store. Recall from the previous sections, for the non-secure case, the following command was executed to produce a JAR file containing only the class files needed by `CountTableRows`.

```
cd /opt/oracle/nosql/apps/kv/examples
jar cvf CountTableRows.jar hadoop/table/CountTableRows*.class
```

which produced the file `CountTableRows.jar`, whose contents look like:

```
META-INF/
META-INF/MANIFEST.MF
hadoop/table/CountTableRows.class
hadoop/table/CountTableRows$Map.class
hadoop/table/CountTableRows$Reduce.class
```

and the following commands were then be used to execute the `CountTableRows` example MapReduce job against a non-secure store:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/ondb/kv/lib/kvclient.jar

cd /opt/ondb/kv
hadoop jar examples/non_secure_CountTableRows.jar \
    hadoop.table.CountTableRows \
    -libjars \
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
```

```
/opt/oracle/kv-ee/lib/jackson-core.jar,\
/opt/oracle/kv-ee/lib/jackson-databind.jar,\
/opt/oracle/kv-ee/lib/jackson-annotations.jar \
example-store \
kv-host-1:5000 \
vehicleTable \
/user/example-user/CountTableRows/vehicleTable/0001
```

Observe that there are three classpaths that must be set when a MapReduce job is executed. First, the jar specification to the Hadoop command interpreter makes the class files of the main program (`CountTableRows` in this case) accessible to the hadoop launcher mechanism, so that the program can be loaded and executed. Next, the `HADOOP_CLASSPATH` environment variable must be set to include any third party libraries that the program or the Hadoop framework, running on the local access node, may need to load. For the example above, only `kvclient.jar` is added to `HADOOP_CLASSPATH`, so that the Hadoop framework's job initiation mechanism on the access node can access `TableInputFormat` and its related classes. Compare this with the specification of the `-libjars` argument which is the third classpath that must be specified. As described below, the `-libjars` argument must include not only `kvclient.jar`, but also a number of other third party libraries that may not be available in the remote Hadoop environment.

The Hadoop command interpreter's `-libjars` argument is used to specify the classpath needed by each MapReduce task executing on the Hadoop cluster's DataNodes. The `-libjars` argument must include all of the libraries needed to run the desired application that are not already available via the Hadoop platform. For the case above, `kvclient.jar`, `sklogger.jar`, `commonutil.jar`, `failureaccess.jar`, `antlr4-runtime-nosql-shaded.jar`, `jackson-core.jar`, `jackson-databind.jar`, and `jackson-annotations.jar` are each specified via the `-libjars` argument so that each MapReduce task can access classes such as, `TableInputSplit` and `TableRecordReader`, as well as the logging related classes and JSON utility classes provided by Oracle NoSQL Database and other support classes that are not generally provided by the Hadoop platform.

Application Packaging and Execution for the Secure Case

Compare the non-secure case described in the previous section with what would be done to run the `CountTableRows` MapReduce job against a secure store. For the secure case, two JAR files are built; one for the classpath on the client side, and one for the classpaths of the DataNodes on the server side. The first JAR file will be added to the client side classpath and includes not only the class files for the application but also the public and private credentials the client will need to interact with the secure store. Including the public and private credentials in the client side JAR file avoids the inconvenience of having to specify that information on the command line.

The second JAR file will be added to the DataNode classpaths on the server side via the `-libjars` argument, and will include only the user's public credentials.

As described in the Deploying a Secure Store appendix, the user's password can be stored in either a clear text password file or an Oracle Wallet. As a result, how the first JAR is generated is dependent on whether a password file or an Oracle Wallet is used.

Application Packaging for the Secure Case Using a Password File

If you wish to execute `CountTableRows` using a password file instead of an Oracle Wallet, and if you have used `KVSecurityCreation` to generate the user's security artifacts in the manner presented in the Deploying a Secure Store appendix, then both the client side and server side JAR files for the `CountTableRows` example application are generated by typing the following on the command line:

```
cd /opt/oracle/nosql/apps/kv/examples
jar cvf CountTableRows-pwdClient.jar \
    hadoop/table/CountTableRows*.class \
    hadoop/table/KVSecurityUtil*.class

cd /tmp
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdClient.jar \
    client.trust
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdClient.jar \
    example-user-client-pwdfile.login
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdClient.jar \
    example-user.passwd

jar cvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdServer.jar /
    client.trust
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdServer.jar \
    example-user-server.login
```

The first four commands above produce the client side JAR file named `CountTableRows-pwdClient.jar`, where the contents of that JAR look like:

```
META-INF/
META-INF/MANIFEST.MF
hadoop/table/CountTableRows.class
hadoop/table/CountTableRows$Map.class
hadoop/table/CountTableRows$Reduce.class
hadoop/table/KVSecurityUtil.class
client.trust
example-user-client-pwdfile.login
example-user.passwd
```

The following files in the above code correspond to security artifacts that should remain private to the client.

```
example-user-client-pwdfile.login
example-user.passwd
```

The last two commands above produce the server side JAR file named `CountTableRows-pwdServer.jar`, with contents that look like:

```
META-INF/
META-INF/MANIFEST.MF
client.trust
example-user-server.login
```

The last two files from the above list correspond to the client's security artifacts that can be shared publicly.

Application Execution for the Secure Case Using a Password File

If you wish to execute the `CountTableRows` MapReduce job against a secure store where a password file rather than an Oracle Wallet is used to store the client application's password, then after packaging the application for password file based execution as described in the previous section, you would then type the following on the command line:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/oracle/kv-ee/kv/lib/kvclient.jar:\
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-pwdServer.jar

cd /opt/oracle/nosql/apps/kv

hadoop jar examples/CountTableRows-pwdClient.jar \
    hadoop.table.CountTableRows \
    -libjars \
    /opt/oracle/kv-ee/kv/lib/kvclient.jar,\
    /opt/oracle/kv-ee/kv/lib/sklogger.jar,\
    /opt/oracle/kv-ee/kv/lib/commonutil.jar,\
    /opt/oracle/kv-ee/kv/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/kv/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-databind.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-annotations.jar,\
    /opt/oracle/nosql/apps/examples/CountTableRows-pwdServer.jar \
    example-store \
    kv-host-1:5000 \
    vehicleTable \
    /user/example-user/CountTableRows/vehicleTable/0001 \
    example-user-client-pwdfile.login \
    example-user-server.login
```

Application Packaging for the Secure Case Using an Oracle Wallet

Rather than using a file in which to store the client's password, you may choose to use an Oracle Wallet to store the password in obfuscated form. When an Oracle Wallet will be used and the `KVSecurityCreation` convenience program was used to generate the wallet based artifacts for `CountTableRows` in the manner presented in the Deploying a Secure Store appendix, then both the client side and server side JAR files

for the wallet based `CountTableRows` example application are generated by typing the following on the command line:

```
cd /opt/oracle/nosql/apps/kv/examples
jar cvf CountTableRows-walletClient.jar \
    hadoop/table/CountTableRows*.class \
    hadoop/table/KVSecurityUtil*.class

cd /tmp
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletClient.jar \
    client.trust
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletClient.jar \
    example-user-client-walletfile.login
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletClient.jar \
    example-user-wallet.dir

jar cvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletServer.jar /
    client.trust
jar uvf \
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletServer.jar \
    example-user-server.login
```

The first four commands above produce the client side JAR file named `CountTableRows-walletClient.jar`, where the contents of that JAR look like:

```
META-INF/
META-INF/MANIFEST.MF
hadoop/table/CountTableRows.class
hadoop/table/CountTableRows$Map.class
hadoop/table/CountTableRows$Reduce.class
hadoop/table/KVSecurityUtil.class
client.trust
example-user-client-wallet.login
example-user-wallet.dir/
example-user-wallet.dir/cwallet.sso
```

Similarly, the last two commands produce the server side JAR file named `CountTableRows-walletServer.jar`, with contents:

```
META-INF/
META-INF/MANIFEST.MF
client.trust
example-user-server.login
```

Application Execution for the Secure Case Using an Oracle Wallet

If you wish to execute the `CountTableRows` MapReduce job against a secure store using an Oracle Wallet to store the client application's password, then after packaging

the application for wallet based execution as described in the previous section, you would type the following on the command line:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:\
    /opt/oracle/kv-ee/kv/lib/kvclient.jar:\
    /opt/oracle/nosql/apps/kv/examples/CountTableRows-walletServer.jar

cd /opt/oracle/nosql/apps/kv

hadoop jar examples/CountTableRows-walletClient.jar \
    hadoop.table.CountTableRows \
    -libjars \
    /opt/oracle/kv-ee/kv/lib/kvclient.jar,\
    /opt/oracle/kv-ee/kv/lib/sklogger.jar,\
    /opt/oracle/kv-ee/kv/lib/commonutil.jar,\
    /opt/oracle/kv-ee/kv/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/kv/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-databind.jar,\
    /opt/oracle/kv-ee/kv/lib/jackson-annotations.jar,\
    /opt/oracle/nosql/apps/examples/CountTableRows-walletServer.jar \
    example-store \
    kv-host-1:5000 \
    vehicleTable \
    /user/example-user/CountTableRows/vehicleTable/0001 \
    example-user-client-walletfile.login \
    example-user-server.login
```

Secure Versus Non-Secure Command Lines

When examining how the application is executed using either a wallet based or a password file based password storage mechanism, you should first notice that, unlike the non-secure case, the `HADOOP_CLASSPATH` and `-libjars` argument have both been augmented with the JAR file that contains only the public credentials for login and trust; that is, either `CountTableRows-pwdServer.jar` or `CountTableRows-walletServer.jar`. Because those JAR files contain only public information, they can be safely transmitted to the server side remote address spaces.

Compare this with the value to which the application's local classpath is set, via the jar directive. Rather than including the application's server based JAR file, the local classpath instead is set to include the application's client based JAR file; either `CountTableRows-pwdClient.jar` or `CountTableRows-walletClient.jar`. The application's client based JAR file includes both the application's public and private credentials. Those JAR files contain security artifacts which should remain private to the application's address space; that is, the client side of the application. As a result, those JAR files must never be included in the `HADOOP_CLASSPATH` or `-libjars` specifications. They should be included only in the client's local classpath.

Finally, the only other difference between the command lines for secure execution and non-secure execution, is the two additional arguments at the end of the argument list for the secure case; specifically, `example-user-server.login` and either `example-user-client-pwdfile.login` or `example-user-client-wallet.login`.

The values of those arguments specify, respectively, the names of the client side and server side login files, whose contents will be retrieved as resources from the corresponding JAR file.

Observe that when you package and execute your MapReduce application in a manner like that shown here, there is no need to specify the username or password file (or wallet) on the command line; as that information is included as part of the client side JAR file. Additionally, the server side JAR file that is transferred from the Hadoop cluster’s access node to the job's DataNodes does not include that private information. This is important because that transferred JAR file will be cached in the file system of each of those DataNodes.

Summary

As the sections above demonstrate, the programming model for MapReduce and Oracle NoSQL Database Security supports (even encourages) the best practices presented in this section for building, packaging, and deploying any given MapReduce job that employs the Oracle NoSQL Database Table API to retrieve and process data in a given Oracle NoSQL Database store, either secure or non-secure. As a result, simply generating separate JAR files a set of JAR files for the secure case, and one for the non-secure case allows deployers to conveniently run the job with or without security.

> **Note:**
>
> This model for separating public and private user credentials will also play an important role when executing Hive queries against table data in a secure store.

Part II
Integration with Apache Hive

Topics

5

Introduction to Integration with Apache Hive

The Integration with Apache Hadoop MapReduce section describes the set of classes provided by Oracle NoSQL Database that support running Hadoop MapReduce jobs against data stored in an Oracle NoSQL Database table. Since a typical Hive query generally results in the execution of a MapReduce job, it was natural for Oracle NoSQL Database to also provide new interfaces and classes which support running Hive queries against such table data.

In addition to describing the core interfaces and classes involved in running a Hive query against data from a table located in a given Oracle NoSQL Database store, the information presented in this section will also present the steps to take to execute a given set of basic Hive queries against example table data contained in such a store, where the store can be either secure or non-secure.

Prerequisites

Before attempting to execute the example that demonstrates the concepts presented in this section, you should first satisfy the following prerequisites:

- Become familiar with Apache Hive 2 and its programming model. Specifically, become familiar with how to write and execute a Hive query.
- Become familiar with Apache Hadoop 3. Specifically, become familiar with how Hive and Hadoop interact.
- Deploy a Hadoop cluster with 3 data nodes running on machines with host names, `dn-host-1`, `dn-host-2`, and `dn-host-3`.
- Become familiar with the Hive Command Line Interface (the Hive CLI), and the Hive Query Language.
- Become familiar with Oracle NoSQL Database (see Introduction to Oracle NoSQL Database in the Concepts Guide) and then install, start, and configure an Oracle NoSQL Database that is network reachable from the nodes of the Hadoop cluster and any Hive clients. The KVHOME of the store that you start should be configured as the directory `/opt/oracle/kv-ee`.
- Deploy a store to 3 machines (real or virtual) with host names, `kv-host-1`, `kv-host-2`, and `kv-host-3`. The store's name should be set to the value `example-store`, and the store's KVROOT should be set to the directories `/u01/nosql/sn1/kvroot` on `kv-host-1`, `/u02/nosql/sn2/kvroot` on `kv-host-2`, and `/u03/nosql/sn3/kvroot` on `kv-host-3`. Finally, an Oracle NoSQL Database admin service, listening on port 5000, should be deployed to each host making up the store.
- Become familiar with the Oracle NoSQL Database Security model and be able to configure the deployed store for secure access (optional).
- If the deployed store is configured for secure access, start the Oracle NoSQL Database Administrative CLI and then follow the steps presented in the Deploying

a Secure Store appendix to securely connect to the store and create a user named `example-user`, along with the appropriate security artifacts (login file, trust file, and either password file or Oracle Wallet).

- Obtain and install the separate distribution containing the Oracle NoSQL Database example code (see Oracle Technology Network). Although you are free to install that package in any location on your system, for simplicity this document assumes the example code is installed under the directory `/opt/oracle/nosql/apps/kv/examples`.
- Become familiar with the supporting Java classes presented in the Integration with Apache Hadoop MapReduce section, and then follow the steps presented in that document to create and populate the table named `vehicleTable` with example data consisting of only primitive data types.
- Become familiar with the `LoadRmvTable` program provided in the Oracle NoSQL Database example distribution, and then follow the steps presented in the Creating and Populating the rmvTable appendix to create and populate a table named `rmvTable` with example data consisting of both primitive and non-primitive data types.
- Become familiar with the `LoadJsonExample` program provided in the Oracle NoSQL Database example distribution, and then follow the steps presented in the Creating and Populating the exampleJsonTable appendix to create and populate a table named `exampleJsonTable` with rows containing valid JSON formatted data (documents).

Using specific values for items such as the KVHOME and KVROOT environment variables, as well as the store name, host names, admin port, and example code location described above should allow you to more easily understand and use the example commands. Combined with the information contained in the Concepts Guide, along with the Administrator's Guide and Security Guide, you should then be able to generalize and extend these examples to your own particular development scenario; substituting the values specific to the given environment where necessary.

Detailed instructions for deploying a non-secure Oracle NoSQL Database store are provided in the Deploying a Non-Secure Store appendix. Similarly, the Deploying a Secure Store appendix provides instructions for deploying a store configured for security.

A Brief Primer on Apache Hive

Paraphrasing wikipedia, Apache Hive is a data warehouse infrastructure built on top of Apache Hadoop that facilitates querying datasets residing in distributed file systems such as the Hadoop Distributed File System (referred to as HDFS) or in compatible file systems. In addition to those built in features, Hive also provides a pluggable programming model that allows you to specify custom interfaces and classes that support querying data residing in data sources such as the Oracle NoSQL Database.

In addition to the Hive infrastructure and its pluggable programming model, Hive also provides a convenient client-side command line interface (the Hive CLI), which allows you to interact with the Hive infrastructure to create a Hive external table and then map it to the data located in remote sources like those just described.

Oracle NoSQL Database provides a set of interfaces and classes that satisfy the Hive programming model so that the Hive Query Language can be used to query data contained in an Oracle NoSQL Database store (either secure or non-secure). The classes that are defined for that purpose are located in the Java package

`oracle.kv.hadoop.hive.table` (see Java API), and consist of the following Hive and Hadoop types:

- A subclass of the Hive class `org.apache.hadoop.hive.ql.metadata.HiveStorageHandler`. The `HiveStorageHandler` is the mechanism (the pluggable interface) Oracle NoSQL Database uses to specify the location of the data that the Hive infrastructure should process, as well as how that data should be processed. The `HiveStorageHandler` consists of the following components:
  - A subclass of the Hadoop MapReduce version 1 class `org.apache.hadoop.mapred.InputFormat`, where `InputFormat` specifies how the associated MapReduce job reads its input data, taken from the Oracle NoSQL Database table.
  - A subclass of the Hadoop MapReduce version 1 class `org.apache.hadoop.mapred.OutputFormat`, where `OutputFormat` specifies how the associated MapReduce job writes its output.
  - A subclass of the Hive class `org.apache.hadoop.hive.serde2.AbstractSerDe`. The `AbstractSerDe` class and its associated subclasses are used to deserialize the table data that is retrieved and sent to the Hive infrastructure and/or the Hadoop MapReduce job for further processing. Although not currently supported, this mechanism can also be used to serialize data input to Hive for writing to an Oracle NoSQL Database table.
  - Metadata hooks for keeping an external catalog in sync with the Hive Metastore component.
  - Rules for setting up the configuration properties on MapReduce jobs run against the data being processed.
  - An implementation of the interface `org.apache.hadoop.hive.ql.metadata.HiveStoragePredicateHandler`. As described in the Predicate Pushdown appendix, the implementation of `HiveStoragePredicateHandler` provided by Oracle NoSQL Database supports the decomposition of a query's WHERE clause (the predicates of the query) into information that can be passed to the database so that some (or even all) of the search processing can be performed in the database itself rather than on the client side of the query.
- A subclass of the Hadoop MapReduce version 1 class `org.apache.hadoop.mapred.RecordReader`, where a `RecordReader` is used to specify how the mapped keys and values are located and retrieved during any MapReduce processing performed while executing a Hive query.
- A subclass of the Hadoop MapReduce version 1 class `org.apache.hadoop.mapred.InputSplit`, where an `InputSplit` is used to represent the data to be processed by an individual Mapper that operates during the MapReduce processing performed as part of executing a Hive query.

See Apache Hadoop API and Hive API for more details.

As described in the following sections, it is through the implementation of the `HiveStorageHandler` provided by Oracle NoSQL Database that the Hive infrastructure obtains access to a given Oracle NoSQL Database store and ultimately the table data on which to run the desired Hive query.

6

Oracle NoSQL Database Hive Integration Classes

To support running Hive queries against data stored in a table of an Oracle NoSQL Database store, the following core classes are employed:

- `oracle.kv.hadoop.hive.table.TableStorageHandler`
- `oracle.kv.hadoop.hive.table.TableHiveInputFormat`
- `oracle.kv.hadoop.hive.table.TableHiveInputSplit`
- `oracle.kv.hadoop.hive.table.TableHiveRecordReader`
- `oracle.kv.hadoop.hive.table.TableSerDe`
- Implementations specific to Oracle NoSQL Database of the Hive `org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector` interface that support deserialization of the primitive and non-primitive data types defined by the Oracle NoSQL Database table API.

For more detail, see oracle.kv.hadoop.hive.table API.

7

Mapping the Hive Data Model to the Oracle NoSQL Database Table Model

As the examples presented here demonstrate, in order to execute a Hive query against data stored in an Oracle NoSQL Database table, a Hive external table must be created with a schema mapped from the schema of the desired Oracle NoSQL Database table. This is accomplished by applying the mapping described here.

The following implementations of the Hive `ObjectInspector` interface are used in the deserialization process to convert the associated data type defined by the Oracle NoSQL Database table model to its corresponding type in the Hive data model. See oracle.kv.hadoop.hive.table.

- `oracle.kv.hadoop.hive.table.TableBinaryObjectInspector`
- `oracle.kv.hadoop.hive.table.TableBooleanObjectInspector`
- `oracle.kv.hadoop.hive.table.TableDoubleObjectInspector`
- `oracle.kv.hadoop.hive.table.TableFloatObjectorInspector`
- `oracle.kv.hadoop.hive.table.TableIntObjectInspector`
- `oracle.kv.hadoop.hive.table.TableLongObjectInspector`
- `oracle.kv.hadoop.hive.table.TableJsonObjectInspector`
- `oracle.kv.hadoop.hive.table.TableNumberObjectInspector`
- `oracle.kv.hadoop.hive.table.TableTimestampObjectInspector`
- `oracle.kv.hadoop.hive.table.TableEnumObjectInspector`
- `oracle.kv.hadoop.hive.table.TableArrayObjectInspector`
- `oracle.kv.hadoop.hive.table.TableMapObjectInspector`
- `oracle.kv.hadoop.hive.table.TableRecordObjectInspector`

The data model defined by the Oracle NoSQL Database (see oracle.kv.table.FieldDef.Type) is mapped to a subset of the types defined by Hive, as shown in the following table. Specifically, when creating a Hive external table so that you can query the data in a given Oracle NoSQL Database table, the Hive table must be created with a schema consistent with the mappings shown in the following table:

Table 7-1 Hive Data Model

Oracle NoSQL Database Type	Hive Type
FieldDef.Type.STRING	STRING
	CHAR
	VARCHAR
FieldDef.Type.JSON	STRING

Table 7-1 (Cont.) Hive Data Model

Oracle NoSQL Database Type	Hive Type
FieldDef.Type.BOOLEAN	BOOLEAN
FieldDef.Type.BINARY	BINARY
FieldDef.Type.FIXED_BINARY	BINARY
	TINYINT
	SMALLINT
FieldDef.Type.INTEGER	INT
FieldDef.Type.LONG	BIGINT
FieldDef.Type.FLOAT	FLOAT
FieldDef.Type.NUMBER	DECIMAL
FieldDef.Type.DOUBLE	DOUBLE
FieldDef.Type.ENUM	STRING
FieldDef.Type.TIMESTAMP	java.sql.TIMESTAMP
	DATE
FieldDef.Type.ARRAY	ARRAY
FieldDef.Type.MAP	MAP<STRING, data_type>
FieldDef.Type.RECORD	STRUCT<col_name : data_type,…>
	UNIONTYPE<data_type,data_type,…>

For more details, see FieldDef.Type.

It is important to understand that when using Hive to query data in an Oracle NoSQL Database table, the schema of the Hive external table you create is dependent on the schema of the corresponding Oracle NoSQL Database table you wish to query. Thus, if you create a Hive external table with a schema that includes a Hive data type that is not mapped from an Oracle NoSQL Database `FieldDef.Type`, then an error will occur when any attempt is made to query the table.

YARN Versus MapReduce Version 1

Hadoop deployments can include two versions of MapReduce. The first version (referred to as MRv1) is the original version of MapReduce; and consists of interfaces and classes from the Java package `org.apache.hadoop.mapred`. The newer version of MapReduce is referred to as YARN (Yet Another Resource Negotiator) or, more generally, MRv2. Mrv2 resides in the package `org.apache.hadoop.mapreduce`. The Oracle NoSQL Database API Hive integration classes addresses the existence of both versions of MapReduce because:

- Hive currently employs MRv1
- Oracle NoSQL Database API Hadoop integration classes employ MRv2
- MRv1 and MRv2 are source incompatible

To support both MRv1 and MRv2, the Oracle NoSQL Database API Hive integration classes have subclassed the MRv1 classes to the appropriate MRv2 classes. In essence, the `InputFormat` from the `org.apache.hadoop.mapred` is a subclass of the `TableHiveInputFormat` from `oracle.kv.hadoop.hive.table`.

> **Note:**
>
> As the Oracle NoSQL Database Hadoop integration classes do not currently support writing data from a MapReduce job into an Oracle NoSQL Database store, the classes specified here for Hive integration do not support queries that modify the contents of a table in a store.

For more details, see oracle.kv.hadoop.hive.table, Apache Hadoop API, and Hive API.

8

Example: Hive Queries On Oracle NoSQL Database Tables

This section presents examples of how Hive can be configured to query data stored in different tables located in an Oracle NoSQL Database store, either non-secure or secure. The Primitive Data Types - The vehicleTable Example contains only primitive Oracle NoSQL Database data types, and is a good place to start when investigating basic Hive queries. The Non-Primitive Data Types - The rmvTable Example contains a mix of primitive and non-primitive data types, and demonstrates how to query more complex data. Finally, the NoSQL JSON Data Type - The exampleJsonTable Example focuses on how to query JSON documents that have been written to an Oracle NoSQL Database.

Before running any of the queries described here, you must take the following initial steps to setup your system for Hive integration with Oracle NoSQL Database:

- Satisfy the necessary prerequisites, see Prerequisites.
- Follow the directions presented in the Deploying a Non-Secure Store appendix or Deploying a Secure Store appendix to deploy either a non-secure or a secure Oracle NoSQL Database store.
- Follow the instructions presented in the `CountTableRows` Support Programs appendix to create and populate a table named `vehicleTable` in the store that you deployed.
- Follow the instructions presented in the Creating and Populating the rmvTable appendix to create and populate a table named `rmvTable` in the store that you deployed.
- Follow the instructions presented in the Creating and Populating the exampleJsonTable appendix to create and populate a table named `exampleJsonTable` in the store that you deployed.
- Follow the instructions presented in the Configuring the Hive Client Environment appendix to configure the Hive client environment so that it has access to the Oracle NoSQL Database libraries needed to query data stored in the Oracle NoSQL Database store you deployed.
- If the store you deployed is configured for secure access, then follow the steps provided in the Hive and Oracle NoSQL Database Security appendix to configure Hive with the environment and artifacts necessary to interact with a secure store.

Once these initial steps are performed, the sections that follow present Hive commands for creating and mapping Hive external tables to the tables you created in the Oracle NoSQL Database, and then demonstrate how to use Hive to query the data stored in those tables.

Note the following general points about the Hive commands that are presented:

- The contents of the Hive commands presented below are displayed on separate lines for readability. In practice, because the Hive command interpreter may

have trouble handling multi-line commands, it is generally best to enter a single, continuous command with no line breaks.

- When executing the command to create a Hive external table, the `oracle.kv.tableName` property is used to indicate to Hive the name of the table in the Oracle NoSQL Database store that will be queried; where the name specified for the Hive table is not required to be the same as name of the corresponding Oracle NoSQL Database table. We used a combination of both in the examples. In the cases where the names are different, we used a name that was descriptive of the scenario.
- If the Oracle NoSQL Database store is configured with multiple administrative hosts, then any subset of the names of those hosts can be included in the value of the `oracle.kv.hosts` property specified in the command; as long as at least one valid administrative host and port is included.
- With respect to the property named `oracle.kv.hadoop.hosts`:
  - That property is currently optional for all systems except the Big Data SQL system.
  - The property will have no effect if specified on a system that does not require it.
  - When the property is specified on a Big Data SQL system or any other system for which the property is required, the property's value must contain the names of all of the data nodes making up the Hadoop cluster. See Big Data SQL User's Guide.

It's important to understand the different scenarios in which each Hive command is executed and how a given command differs in each scenario. This is because the command used to create a Hive external table mapped to an Oracle NoSQL Database table requires different parameters, where the parameters specified depend on which of the following conditions are met:

- The Oracle NoSQL Database store is non-secure.
- The Oracle NoSQL Database store is secure and your Hive client's password is:
  - Stored in a password file.
  - Stored in an Oracle Wallet.

To understand the difference between the non-secure scenario and the secure scenarios, it will help to compare the command used to map a Hive external table to a table in a non-secure store with the commands used to map two separate Hive tables to a single table in a secure store.

The respective commands in each scenario of a given example will apply the same Hive data model mapping, specified in Table 7-1, to create three different Hive external tables. Each table will have the same structure, schema, and attributes.

The only difference between the table created in the non-secure scenario, and the two tables created in the secure scenario, is the value specified for the Hive table name (for example, `vehicleTable`, `vehicleTablePasswd`, and `vehicleTableWallet`), and whether or not security artifacts needed for communication with a secure store are required to create the desired Hive table.

Specifically, when creating and mapping a Hive external table to a table in a secure Oracle NoSQL Database store, the `TBLPROPERTIES` directive of the Hive `CREATE`

EXTERNAL TABLE command requires that you specify the following additional security-related properties:

- oracle.kv.security
- oracle.kv.auth.username
- oracle.kv.auth.pwdfile.file or oracle.kv.auth.wallet.dir

Each of the properties listed above corresponds to one of the artifacts Oracle NoSQL Database requires for Hive to securely communicate with the store identified by the remaining properties specified in the Hive TBLPROPERTIES directive.

For details on the nature of each of the additional security related properties, refer to the Model For Building & Packaging Secure Clients appendix.

Other than the differences just described, with respect to the Hive commands presented in the following sections, the non-secure scenario and the secure scenario are the same in all other aspects.

Primitive Data Types - The vehicleTable Example

This example demonstrates how to execute various Hive queries on a simple Oracle NoSQL Database table containing only primitive data types.

The Hive queries executed in this example will be applied to the table named vehicleTable you initially created and populated in the Oracle NoSQL Database store. For more information on that table's schema and data types, see the CountTableRows Support Programs appendix.

Prior to executing Hive queries against the Oracle NoSQL Database vehicleTable, you must first create an external table in Hive and map it to the table in Oracle NoSQL Database.

Mapping a Hive External Table to vehicleTable: Non-Secure Store

Assuming you have executed the initial steps to deploy a non-secure store, created and populated the table named vehicleTable in that store, and configured the Hive client environment for interaction with Oracle NoSQL Database, you can then create an external Hive table that maps to that Oracle NoSQL Database table by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTable
    (type STRING, make STRING, model STRING, class STRING, color STRING,
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES ("oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts" = "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000"
        "oracle.kv.tableName" = "vehicleTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

The command above applies the required data model mapping to create a Hive table named vehicleTable with columns whose types are consistent with the corresponding fields of the Oracle NoSQL Database table specified via the oracle.kv.tableName property.

> **Note:**
>
> Although not necessary, the Hive table that is created is given the same name as the table to which it is mapped in the store.

Mapping a Hive External Table to vehicleTable: Secure Store

Assuming you have executed the initial steps to deploy a secure store, created and populated the table named `vehicleTable` in that store, and configured the Hive client environment for secure interaction with Oracle NoSQL Database, you can then create two external Hive tables that each map to that single Oracle NoSQL Database table by executing the Hive commands presented in both of the following sections.

When mapping a Hive external table to a table located in a secure Oracle NoSQL Database, because the password the Hive client uses to access and communicate with the store can be stored in either a password file or an Oracle Wallet, the following sections present commands that take different parameters, depending on the mechanism used to store the user's password.

Mapping Hive to Secure vehicleTable: Password File

If a password file is used for password storage, then you can create an external Hive table that maps to the `vehicleTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTablePasswd
    (type STRING, make STRING, model STRING, class STRING, color STRING,
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES ("oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "vehicleTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

Mapping Hive to Secure vehicleTable: Oracle Wallet

If an Oracle Wallet is used for password storage, then you can create an external Hive table that maps to the `vehicleTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTableWallet
    (type STRING, make STRING, model STRING, class STRING, color
STRING,
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES ("oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
```

```
"oracle.kv.tableName" = "vehicleTable",
"oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
"oracle.kv.security" = "/tmp/hive-nosql.login",
"oracle.kv.ssl.truststore" = "/tmp/client.trust",
"oracle.kv.auth.username" = "example-user",
"oracle.kv.auth.wallet.dir" = "/tmp/example-user-wallet.dir");
```

Hive Queries on vehicleTable: Primitive Data Types

After following the directions presented in the previous sections to create and map a Hive external table to the `vehicleTable` in the Oracle NoSQL Database store (either non-secure or secure), that table can be queried via the Hive Query Language.

In the previous sections, three scenarios were presented for mapping a Hive table to a table in a given Oracle NoSQL Database store: a non-secure store, a secure store in which the client store's its password in a password file, and a secure store in which the client store's its password in an Oracle Wallet. As a result, each of the following sections present three forms of a given query, one for each scenario; specifically,

- A query on the table named `vehicleTable` in the non-secure scenario
- A query on `vehicleTablePasswd` in the secure (with password file) scenario
- A query on `vehicleTableWallet` in the secure (with wallet) scenario

The only difference between a given query from scenario to scenario is in the name of the table to query.

Because the Hive table created for each separate scenario is mapped to the same underlying Oracle NoSQL Database table, the output of each form of a given query for each scenario will be the same. Thus, although each section presents three instances of a given query, the query result is shown only once, and is edited for clarity.

At the Hive CLI command prompt, type the query from each section below that corresponds to how you have configured your particular environment; non-secure or secure store and, if secure, whether you are using a password file or an Oracle Wallet to store the client's password.

> **Note:**
>
> In some cases Hive will execute a MapReduce job to satisfy the query, whereas in other cases, the query is satisfied by simply consulting the Hive data dictionary and so MapReduce is not employed.

List Each Row in the Oracle NoSQL Database vehicleTable

```
SELECT * FROM vehicleTable;
SELECT * FROM vehicleTablePasswd;
SELECT * FROM vehicleTableWallet;

OK
auto  Ford Focus  4WheelDrive white  20743.94 15 3 2020-10-09
auto  GM   Impala 4WheelDrive black  29834.91 24 7 2019-12-11
auto  GM   Impala 4WheelDrive yellow 21753.53 27 8 2017-03-31
```

```
truck Ford F250   4WheelDrive blue   31115.76 14 9 2018-02-01
..........
```

Count the Rows in vehicleTable

```
SELECT count(type) FROM vehicleTable;
SELECT count(type) FROM vehicleTablePasswd;
SELECT count(type) FROM vehicleTableWallet;

OK
Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 0%,  reduce = 0%
Stage-1 map = 7%,  reduce = 0%, Cumulative CPU 2.26 sec
Stage-1 map = 21%,  reduce = 0%, Cumulative CPU 6.7 sec
Stage-1 map = 30%,  reduce = 0%, Cumulative CPU 6.87 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 14.16 sec
Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 15.24 sec
..........
Job 0: Map: 6  Reduce: 1   Cumulative CPU: 15.24 sec
HDFS Read: 4532 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 15 seconds 240 msec
OK
79
Time taken: 89.359 seconds, Fetched: 1 row(s)
```

Find the Lowest Price On Any Vehicle in vehicleTable

```
SELECT min(price) FROM vehicleTable;
SELECT min(price) FROM vehicleTablePasswd;
SELECT min(price) FROM vehicleTableWallet;

OK
Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 0%,  reduce = 0%
Stage-1 map = 21%,  reduce = 0%, Cumulative CPU 6.7 sec
Stage-1 map = 21%,  reduce = 0%, Cumulative CPU 6.7 sec
Stage-1 map = 30%,  reduce = 0%, Cumulative CPU 6.87 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 12.16 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 14.16 sec
..........
Job 0: Map: 6  Reduce: 1   Cumulative CPU: 15.24 sec
HDFS Read: 4532 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 15 seconds 240 msec
OK
20743.94
Time taken: 89.615 seconds, Fetched: 1 row(s)
```

List All GM Vehicles in vehicleTable

```
SELECT * FROM vehicleTable WHERE make LIKE "%GM";
SELECT * FROM vehicleTablePasswd WHERE make LIKE "%GM";
SELECT * FROM vehicleTableWallet WHERE make LIKE "%GM";

OK
Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 0
Stage-1 map = 0%,  reduce = 0%
Stage-1 map = 9%,  reduce = 0%, Cumulative CPU 2.43 sec
Stage-1 map = 26%,  reduce = 0%, Cumulative CPU 4.81 sec
Stage-1 map = 79%,  reduce = 0%, Cumulative CPU 13.09 sec
Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 16.06 sec
..........
Job 0: Map: 6  Cumulative CPU: 15.24 sec
HDFS Read: 4532 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 15 seconds 240 msec
OK

suv   GM Equinox  4WheelDrive   white  20743.94 3 1 2019-03-01
truck GM Sierra   4WheelDrive   black  29834.91 8 3 2020-05-15
auto  GM Corvette 4WheelDrive   yellow 21753.53 7 5 2017-10-23
auto  GM Impala   AllWheelDrive blue   31115.76 4 9 2018-05-04
..........
Time taken: 89.615 seconds, Fetched: 1 row(s)
```

List All GM Vehicles in vehicleTable that are Red or Blue

```
SELECT * FROM vehicleTable WHERE color IN ('red','blue') AND make='GM';
SELECT * FROM vehicleTablePasswd WHERE color IN ('red','blue') AND
make='GM';
SELECT * FROM vehicleTableWallet WHERE color IN ('red','blue') AND
make='GM';

OK
auto  GM Tahoe   AllWheelDrive  red  20743.67 28 3 2019-02-04
auto  GM Sierra  RearWheelDrive blue 20744.10 63 7 2018-08-04
suv   GM Tahoe   RearWheelDrive red  41486.74 27 5 2020-07-08
truck GM Equinox 4WheelDrive    red  31115.17 31 9 2017-06-04
truck GM Blazer  AllWheelDrive  red  31114.83 69 2 2019-04-06
truck GM Sierra  4WheelDrive    blue 31115.32 85 7 2019-02-11
..........
```

Query a Range of Delivery Times and Order the Results

```
SELECT delivered FROM vehicleTable WHERE delivered
    BETWEEN '2020-06-06 06:53:41.448643' AND '2019-09-05
15:40:22.057282'
    ORDER BY delivered;
SELECT delivered FROM vehicleTablePasswd WHERE delivered
```

```
    BETWEEN '2020-06-06 06:53:41.448643' AND '2019-09-05
15:40:22.057282'
    ORDER BY delivered;
SELECT delivered FROM vehicleTableWallet WHERE delivered
    BETWEEN '2020-06-06 06:53:41.448643' AND '2019-09-05
15:40:22.057282'
    ORDER BY delivered;

OK
2019-07-25 09:19:24.743
2019-10-24 00:13:29.702
2019-11-04 05:31:34.358
..........
2020-05-25 06:15:16.809
2020-07-05 12:16:03.658
2020-09-05 15:40:22.057
```

Non-Primitive Data Types - The rmvTable Example

This example demonstrates how to execute various Hive queries on an Oracle NoSQL Database table defined with a complex schema. In this example, a schema is employed that consists of a variety of Oracle NoSQL Database data types; both primitive and non-primitive.

The Hive queries executed in this example will be applied to the table named `rmvTable` you initially created and populated in the Oracle NoSQL Database store. For more information on that table's schema and data types, see the Creating and Populating the rmvTable appendix.

Prior to executing Hive queries against `rmvTable`, you must first create an external table in Hive and map it to the table in Oracle NoSQL Database, as shown in the following sections.

Mapping a Hive External Table to rmvTable: Non-Secure Store

Assuming you have executed the initial steps to deploy a non-secure store, created and populated the table named `rmvTable` in that store, and configured the Hive client environment for interaction with Oracle NoSQL Database, you can then create an external Hive table that maps to that Oracle NoSQL Database table by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTable
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT, street:STRING,
            unit:INT, city:STRING, zip:INT>,
        vehicleinfo ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES
        ("oracle.kv.kvstore" = "example-store",
            "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
```

```
host-3:5000",
            "oracle.kv.tableName" = "rmvTable",
            "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

The command above applies the required data model mapping to create a Hive table named `rmvTable` with columns whose types are consistent with the corresponding fields of the Oracle NoSQL Database table specified via the `oracle.kv.tableName` property.

Although not necessary, the Hive table that is created is given the same name as the table to which it is mapped in the store.

Mapping a Hive External Table to rmvTable: Secure Store

Assuming you have executed the initial steps to deploy a secure store, created and populated the table named `rmvTable` in that store, and configured the Hive client environment for secure interaction with Oracle NoSQL Database, you can then create two external Hive tables that each map to that Oracle NoSQL Database table by executing the Hive commands presented in the following sections.

When mapping a Hive external table to a table located in a secure Oracle NoSQL Database, because the password the Hive client uses to access and communicate with the store can be stored in either a password file or an Oracle Wallet, the sections below present commands that take different parameters, depending on the mechanism used to store the user's password.

Mapping Hive to Secure rmvTable: Password File

If a password file is used for password storage, then you can create an external Hive table that maps to the `rmvTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTablePasswd
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT street:STRING,
            unit:INT, city:STRING, zip:INT>,
        vehicleinfo ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES
        ("oracle.kv.kvstore" = "example-store",
            "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
            "oracle.kv.tableName" = "rmvTable",
            "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
            "oracle.kv.security" = "/tmp/hive-nosql.login",
            "oracle.kv.ssl.truststore" = "/tmp/client.trust",
            "oracle.kv.auth.username" = "example-user",
            "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

Mapping Hive to Secure rmvTable: Oracle Wallet

If an Oracle Wallet is used for password storage, then you can create an external Hive table that maps to the `rmvTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTableWallet
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT street:STRING,
            unit:INT, city:STRING, zip:INT>,
        vehicleinfo ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES
        ("oracle.kv.kvstore" = "example-store",
            "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
            "oracle.kv.tableName" = "rmvTable",
            "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
            "oracle.kv.security" = "/tmp/hive-nosql.login",
            "oracle.kv.ssl.truststore" = "/tmp/client.trust",
            "oracle.kv.auth.username" = "example-user",
            "oracle.kv.auth.wallet.dir" = "/tmp/example-user-
wallet.dir");
```

Hive Queries on rmvTable: Non-Primitive Data Types

After following the directions presented in the previous sections to create and map a Hive external table to a table in the Oracle NoSQL Database store (either non-secure or secure), the data in the store's table can be queried via the Hive Query Language.

In a fashion similar to the `vehicleTable` queries presented previously, each sub-section below presents three instances of a given query, one for each of the three possible scenarios: non-secure, secure with password file, or secure with Oracle Wallet. But the query results are shown only once, in edited form.

Type the query from each sub-section below that corresponds to how you have configured your particular environment.

List Each Row in the rmvTable Located in Oracle NoSQL Database

```
SELECT * FROM rmvTable;
SELECT * FROM rmvTablePasswd;
SELECT * FROM rmvTableWallet;

OK
49027  GOMEZ CHRISTOPHER  509367447  male  S57428836
{"cell":"616-351-0185","home":"213-630-2419","work":"617-227-9840"}
{
    "number":88072,
    "street":"Fifth Avenue",
```

```
    "unit":6,
    "city":"Cambridge",
    "state":"OK",
    "zip":49027
}
[
    {
        "type":"auto",
        "make":"Ford",
        "model":"Taurus",
        "class":"AllWheelDrive",
        "color":"blue",
        "value":20743.234,
        "tax":566.29,
        "paid":false
    },
    {
        "type":"auto",
        "make":"Ford",
        "model":"Taurus",
        "class":"FrontWheelDrive",
        "color":"blue",
        "value":20743.559,
        "tax":566.29,
        "paid":true
    }
]

40719  ROSARIO  ANNE  448406765  female  S04809975
{"cell":"303-804-1660","home":"408-630-2412","work":"415-804-9515"}
{
    "number":96581,
    "street":"Third Avenue",
    "unit":7,
    "city":"Springfield",
    "state":"RI",
    "zip":40719
}
[
    {
        "type":"truck",
        "make":"Chrysler",
        "model":"Ram3500",
        "class":"RearWheelDrive",
        "color":"blue",
        "value":31115.26,
        "tax":849.44,
        "paid":true
    },
    {
        "type":"truck",
        "make":"Chrysler",
        "model":"Ram1500",
        "class":"AllWheelDrive",
        "color":"blue",
```

```
        "value":31114.87,
        "tax":849.43,
        "paid":false
    },
    {
        "type":"auto",
        "make":"Ford",
        "model":"Edge",
        "class":"RearWheelDrive",
        "color":"yellow",
        "value":20743.88,
        "tax":566.30,
        "paid":true
    }
]
..........
```

List Name, Gender, and Address of Each Vehicle Owner in rmvTable

```
SELECT lastname,firstname,gender,address FROM rmvTable;
SELECT lastname,firstname,gender,address FROM rmvTablePasswd;
SELECT lastname,firstname,gender,address FROM rmvTableWallet;

Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 0%,  reduce = 0%
Stage-1 map = 7%,  reduce = 0%, Cumulative CPU 2.26 sec
Stage-1 map = 80%,  reduce = 0%, Cumulative CPU 6.87 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 14.16 sec
..........
Job 0: Map: 6  Reduce: 1   Cumulative CPU: 14.16 sec
HDFS Read: 4760 HDFS Write: 4702 SUCCESS
Total MapReduce CPU Time Spent: 14 seconds 160 msec
OK
SNIDER FRANK  male
{
    "number":33512,
    "street":"Summer Street",
    "unit":1,
    "city":"Arlington",
    "state":"TN","zip":89150
}

MILLER ROCH  male
{
    "number":25698,
    "street":"Mullberry Street",
    "unit":6,
    "city":"Madison",
    "state":"VA",
    "zip":5740
}
```

```
TATE BENJAMIN  male
{
    "number":2894,
    "street":"View Street",
    "unit":-1,
    "city":"Clinton",
    "state":"KY",
    "zip":57466
}
  ..........
```

List Name and Phone Number of Each Vehicle Owner in rmvTable

```
SELECT firstname,lastname,phoneinfo["home"] FROM rmvTable;
SELECT firstname,lastname,phoneinfo["cell"] FROM rmvTablePasswd;
SELECT firstname,lastname,phoneinfo["work"] FROM rmvTableWallet;

Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 0%,  reduce = 0%
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 18.11 sec
..........
Job 0: Map: 6   Cumulative CPU: 18.11 sec
HDFS Read: 4724 HDFS Write: 2141 SUCCESS
Total MapReduce CPU Time Spent: 18 seconds 110 msec
OK

CHRISTOPHER GOMEZ  213-630-2419
ANNE ROSARIO  408-630-2412
MEGAN PHELPS  978-541-5710
MICHAEL BRADLEY  313-351-4580
..........
```

Count Total Number of Rows in rmvTable

```
SELECT count(vehicleinfo[0].type) FROM rmvTable;
SELECT count(vehicleinfo[0].type) FROM rmvTablePasswd;
SELECT count(vehicleinfo[0].type) FROM rmvTableWallet;

Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 50%,  reduce = 0%, Cumulative CPU 12.12 sec
Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 25.51 sec
..........
Job 0: Map: 6  Reduce: 1   Cumulative CPU: 25.51 sec
HDFS Read: 4760 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 25 seconds 510 msec
```

```
OK
79
```

For Each Owner's Primary Vehicle, Find the Minimum Assessed Value

```
SELECT min(vehicleinfo[0].value) FROM rmvTable;
SELECT min(vehicleinfo[0].value) FROM rmvTablePasswd;
SELECT min(vehicleinfo[0].value) FROM rmvTableWallet;

Launching Job 1 out of 1
.........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 14.3 sec
Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 15.38 sec
.........
Job 0: Map: 6  Reduce: 1   Cumulative CPU: 15.38 sec
HDFS Read: 4532 HDFS Write: 16 SUCCESS
Total MapReduce CPU Time Spent: 15 seconds 380 msec
OK
20743.24
```

List All Info For Each Owner's Vehicle (Primary, Secondary, Tertiary)

```
SELECT vehicleinfo[0] FROM rmvTable;
SELECT vehicleinfo[1] FROM rmvTablePasswd;
SELECT vehicleinfo[2] FROM rmvTableWallet;

Launching Job 1 out of 1
.........
Hadoop job information for Stage-1: number of mappers: 6;
                                                number of reducers: 1
Stage-1 map = 17%,  reduce = 0%, Cumulative CPU 4.59 sec
Stage-1 map = 95%,  reduce = 0%, Cumulative CPU 27.33 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 27.89 sec
.........
Job 0: Map: 6   Cumulative CPU: 27.89 sec
                            HDFS Read: 4760 HDFS Write: 5681 SUCCESS
Total MapReduce CPU Time Spent: 27 seconds 890 msec
OK
{
    "type":"suv",
    "make":"GM",
    "model":"Tahoe",
    "class":"4WheelDrive",
    "color":"black",
    "value":41487.24,
    "tax":1132.60,
    "paid":true
}
{
    "type":"auto",
    "make":"Chrysler",
    "model":"Imperial",
```

```
    "class":"AllWheelDrive",
    "color":"red",
    "value":20743.92,
    "tax":566.30,
    "paid":true
}
{
    "type":"auto",
    "make":"Ford",
    "model":"Taurus",
    "class":"RearWheelDrive",
    "color":"blue",
    "value":20744.07,
    "tax":566.31,
    "paid":true
}
..........
```

List Name, Address, Vehicle Info For Owner Surnames Starting With 'H'

```
SELECT firstname,lastname,address,vehicleinfo[0] FROM rmvTable
    WHERE RLIKE "^[H].*";

SELECT firstname,lastname,address,vehicleinfo[0]FROM rmvTablePasswd
    WHERE RLIKE "^[H].*";

SELECT firstname,lastname,address,vehicleinfo[0] FROM rmvTableWallet
    WHERE RLIKE "^[H].*";

Launching Job 1 out of 1
..........
Hadoop job information for Stage-1: number of mappers: 6;
number of reducers: 1
Stage-1 map = 33%,  reduce = 0%, Cumulative CPU 9.46 sec
Stage-1 map = 83%,  reduce = 0%, Cumulative CPU 23.29 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 27.78 sec
..........
Job 0: Map: 6   Cumulative CPU: 27.78 sec
HDFS Read: 4760 HDFS Write: 1143 SUCCESS
Total MapReduce CPU Time Spent: 27 seconds 780 msec
OK

CINDY HODGES
{
    "number":56758,
    "street":"Vaughan Avenue",
    "unit":-1,
    "city":"Madison",
    "state":"NH",
    "zip":79623
}
{
    "type":"truck",
    "make":"Chrysler",
```

```
    "model":"Ram1500",
    "class":"RearWheelDrive",
    "color":"black",
    "value":31115.12,
    "tax":849.44,
    "paid":true
}
JULIA HOLDEN
{
    "number":56209,
    "street":"Main Street",
    "unit":1,"city":"Georgetown",
    "state":"CA",
    "zip":62154
}
{
    "type":"auto",
    "make":"Ford",
    "model":"Taurus",
    "class":"FrontWheelDrive",
    "color":"blue",
    "value":20743.80,
    "tax":566.30,
    "paid":true
}
..........
```

List Name, Address, Vehicle Info When Owner's Second Vehicle Is GM

```
SELECT firstname,lastname,address,vehicleinfo[1] FROM rmvTable
    WHERE vehicleinfo[1].make LIKE "%GM%";

SELECT firstname,lastname,address,vehicleinfo[1] FROM rmvTablePasswd
    WHERE vehicleinfo[1].make LIKE "%GM%";

SELECT firstname,lastname,address,vehicleinfo[1] FROM rmvTableWallet
    WHERE vehicleinfo[1].make LIKE "%GM%";

   Launching Job 1 out of 1
   ..........
   Hadoop job information for Stage-1: number of mappers: 6;
                                                   number of reducers: 1
   Stage-1 map = 50%,  reduce = 0%, Cumulative CPU 9.29 sec
   Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 18.8 sec
   ..........
   Job 0: Map: 6   Cumulative CPU: 18.8 sec
                               HDFS Read: 4724 HDFS Write: 2087 SUCCESS
   Total MapReduce CPU Time Spent: 18 seconds 800 msec
   OK

   NANCY STOUT
   {
     "number":31126,
     "street":"Cedar Street",
```

```
    "unit":8,
    "city":"Arlington",
    "state":"MO",
    "zip":73131
  }
  {
    "type":"suv",
    "make":"GM",
    "model":"Equinox",
    "class":"AllWheelDrive",
    "color":"red",
    "value":41486.43,
    "tax":1132.57,
    "paid":true
  }
  RANDY MCDOWELL
  {
    "number":18391,
    "street":"Lane Avenue",
    "unit":8,
    "city":"Concord",
    "state":"NH",
    "zip":42540
  }
  {
    "type":"auto",
    "make":"GM",
    "model":"Corvette",
    "class":"FrontWheelDrive",
    "color":"black",
    "value":20744.03,
    "tax":566.31,
    "paid":false
  }
  ..........
```

List Name, Address, Model, Assessed Value & Registration Fee Status (Paid or Not) When Primary Vehicle Is Chrysler

```
SELECT firstname, lastname, address, vehicleInfo[0].model,
       vehicleinfo[0].value, vehicleinfo[0].tax, vehicleinfo[0].paid
   FROM rmvTable WHERE vehicleinfo[0]make LIKE "%Chrysler%";

SELECT firstname,lastname,address, vehicleInfo[0].model,
       vehicleinfo[0].value, vehicleinfo[0].tax, vehicleinfo[0].paid
   FROM rmvTablePasswd WHERE vehicleinfo[0]make LIKE "%Chrysler%";

SELECT firstname,lastname,address, vehicleInfo[0].model,
       vehicleinfo[0].value, vehicleinfo[0].tax, vehicleinfo[0].paid
   FROM rmvTableWallet WHERE vehicleinfo[0]make LIKE "%Chrysler%";

   Launching Job 1 out of 1
   ..........
   Hadoop job information for Stage-1: number of mappers: 6;
```

```
                                                  number of reducers: 1
Stage-1 map = 43%,  reduce = 0%, Cumulative CPU 9.46 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 18.15 sec
..........
Job 0: Map: 6   Cumulative CPU: 18.15 sec
                            HDFS Read: 4724 HDFS Write: 2164 SUCCESS
Total MapReduce CPU Time Spent: 18 seconds 150 msec
OK

ANNE ROSARIO
{
  "number":96581,
  "street":"Third Avenue",
  "unit":7,
  "city":"Springfield",
  "state":"RI","zip":40719
}
Ram3500 31115.26 849.44 true

MEGAN PHELPS
{
  "number":12713,
  "street":"MAC Avenue",
  "unit":4,"city":"Salem",
  "state":"MS",
  "zip":76554
}
Ram1500 31115.30 849.44 true

BRIAN ROWLAND
{
  "number":37868,
  "street":"First Street",
  "unit":3,
  "city":"Salem",
  "state":"GA",
  "zip":98106
}
Imperial 20744.15 566.31 true
..........
```

NoSQL JSON Data Type - The exampleJsonTable Example

The `exampleJsonTable` is used to demonstrate Hive queries on an Oracle NoSQL Database table in which one of the table's fields (columns) contains text in valid JSON format; that is, a JSON document. For this example table, a schema is employed that consists of only two fields: a field of type `FieldDef.Type.INTEGER` representing a unique identifier used for the primary key, and a field of type `FieldDef.Type.JSON`, in which each row in the field contains a JSON document consisting of attributes corresponding to information about current and former members of the United States senate; for example, a given senator's name, address, birthday, etc.

The Creating and Populating the exampleJsonTable appendix describes the `exampleJsonTable` in more detail. That appendix also presents a program that

can be run to create and populate `exampleJsonTable` with the sort of data the queries presented in this section expect. Before proceeding, please follow the directions provided in that appendix to create and populate `exampleJsonTable` with the appropriate example data. Then, prior to executing any Hive queries against `exampleJsonTable`, first create a Hive external table and map it to the table you created in Oracle NoSQL Database, as shown in the following sections.

Mapping a Hive External Table to exampleJsonTable: Non-Secure Store

Assuming you have executed the initial steps to deploy a non-secure store, followed the directions presented in the Creating and Populating the exampleJsonTable appendix to create and populate the Oracle NoSQL Database table named `exampleJsonTable`, and configured the Hive client environment for interaction with Oracle NoSQL Database, you can then create an external Hive table that maps to that Oracle NoSQL Database table by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleJsonTable
    (id INT, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

The command above applies the required data model mapping to create a Hive table named `exampleJsonTable` with columns whose types are consistent with the corresponding fields of the Oracle NoSQL Database table specified via the `oracle.kv.tableName` property.

Although not necessary, the Hive table that is created is given the same name as the table to which it is mapped in the store.

Mapping a Hive External Table to exampleJsonTable: Secure Store

Assuming you have executed the initial steps to deploy a secure store, followed the directions presented in the Creating and Populating the exampleJsonTable appendix to create and populate the NoSQL table named `exampleJsonTable`, and configured the Hive client environment for secure interaction with Oracle NoSQL Database, you can then create two external Hive tables that each map to that Oracle NoSQL Database table by executing the Hive commands presented in the following sections.

When mapping a Hive external table to a table located in a secure Oracle NoSQL Database, because the password the Hive client uses to access and communicate with the store can be stored in either a password file or an Oracle Wallet, the sections below present commands that take different parameters, depending on the mechanism used to store the user's password.

Mapping Hive to Secure exampleJsonTable: Password File

If a password file is used for password storage, then you can create an external Hive table that maps to the `exampleJsonTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleTablePasswd
    (id INTEGER, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"="kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

Mapping Hive to Secure exampleJsonTable: Oracle Wallet

If an Oracle Wallet is used for password storage, then you can create an external Hive table that maps to the `exampleJsonTable` by executing the following Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleJsonTableWallet
    (int INTEGER, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"="kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.wallet.dir" = "/tmp/example-user-wallet.dir");
```

Hive Queries on exampleJsonTable: JSON Data Type

After following the directions presented in the previous sections to create and map a Hive external table to the `exampleJsonTable` table in the Oracle NoSQL Database store (either non-secure or secure), the data in the store's table can be queried via the Hive Query Language.

Each sub-section below presents three instances of a given query, one for each of the three possible scenarios: non-secure, secure with password file, or secure with Oracle Wallet. But the query results are shown only once, in edited form.

Type the query from each sub-section below that corresponds to how you have configured your particular environment.

List Each Senator, Ordered By Rank and State

```
SELECT
    get_json_object(jsonfield, '$.description')
        AS description,
    get_json_object(jsonfield, '$.personal.firstname')
        AS firstname,
    get_json_object(jsonfield, '$.personal.lastname')
        AS lastname
FROM exampleJsonTable ORDER BY description;

.........
Hadoop job information for Stage-1: number of mappers: 2;
number of reducers: 1
Stage-1 map = 50%,  reduce = 0%, Cumulative CPU 16.29 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 28.8 sec
.........
Job 0: Map: 2   Cumulative CPU: 34.22 sec
HDFS Read: 16799 HDFS Write: 5490 SUCCESS
Total MapReduce CPU Time Spent: 34 seconds 220 msec
.........
OK

Junior Senator for Alabama       Doug    Jones
Junior Senator for Alaska        Dan     Sullivan
Junior Senator for Arizona       Jeff    Flake
Junior Senator for Arkansas      Tom     Cotton
Junior Senator for California    Kamala  Harris
Junior Senator for Colorado      Cory    Gardner
.........
Senior Senator for Virginia      Mark    Warner
Senior Senator for Washington    Patty   Murray
Senior Senator for West Virginia Joe     Manchin
Senior Senator for Wisconsin     Ron     Johnson
Senior Senator for Wyoming       Michael Enzi

Time taken: 29.342 seconds, Fetched: 100 row(s)
```

> **Note:**
>
> In the SELECT query, instead of the `exampleJsonTable`, you could use any of `exampleJsonTable`, `exampleJsonTablePasswd`, or `exampleJsonTableWallet`.

List Each Senator Who Is An Independent

```
SELECT get_json_object(jsonfield, '$. personal.firstname'),
       get_json_object(jsonfield, '$. personal.lastname'),
```

```
       get_json_object(jsonfield, '$.party'),
       get_json_object(jsonfield, '$.description')
FROM exampleJsonTable ORDER BY description;

.........
Hadoop job information for Stage-1: number of mappers: 2;
number of reducers: 1
Stage-1 map = 50%,  reduce = 0%, Cumulative CPU 11.29 sec
Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 19.67 sec
.........
Job 0: Map: 2   Cumulative CPU: 19.67 sec
HDFS Read: 13716 HDFS Write: 301 SUCCESS
Total MapReduce CPU Time Spent: 19 seconds 670 msec
.........
OK

Angus   King    Independent Junior Senator for Maine
Bernard Sanders Independent Junior Senator for Vermont

Time taken: 15.614 seconds, Fetched: 2 row(s)
```

Note:

In the SELECT query, instead of the `exampleJsonTable`, you could use any of `exampleJsonTable`, `exampleJsonTablePasswd`, or `exampleJsonTableWallet`.

9
Appendix

Topics

Creating and Populating the rmvTable

Oracle NoSQL Database provides a separate distribution consisting of example programs and utility classes that you can use to explore various aspects of interacting with an Oracle NoSQL Database system. With respect to exploring the integration of Oracle NoSQL Database with Apache Hive, that separate example distribution provides a number of Java programs that you can use to create and populate example tables in the Oracle NoSQL Database store you deploy.

The first program is named `LoadTableVehicle` and is described in detail in the `CountTableRows` Support Programs appendix. A second program is named `LoadRmvTable`, and is described in this section, along with the schema employed when creating the table, as well as how to compile and execute the program.

Schema for the Example Table Named rmvTable

To demonstrate how a Hive query can be applied to an Oracle NoSQL Database table consisting of a mix of primitive and non-primitive data types, a table named `rmvTable` having the schema shown in the table below must be created in the Oracle NoSQL Database store deployed for this example. The data types specified in the schema shown below are defined by the Oracle NoSQL Database table API (see FieldDef.Type) .

Table 9-1 Schema for rmvTable

Field Name	Field Type
zipcode	FieldDef.Type.STRING
lastname	FieldDef.Type.STRING
first name	FieldDef.Type.STRING
ssn	FieldDef.Type.STRING
gender	FieldDef.Type.ENUM
license	FieldDef.Type.FIXED_BINARY(9)
phoneinfo	FieldDef.Type.MAP(STRING)
address	FieldDef.Type.RECORD

Table 9-1 (Cont.) Schema for rmvTable

Field Name		Field Type
address Record Schema	number	FieldDef.Type.INTEGER
	street	FieldDef.Type.STRING
	unit	FieldDef.Type.INTEGER
	city	FieldDef.Type.STRING
	state	FieldDef.Type.STRING
	zip	FieldDef.Type.INTEGER
vehicleinfo		FieldDef.Type.ARRAY (FieldDef.Type.RECORD)
vehicleinfo Element Record Schema	type	FieldDef.Type.STRING
	make	FieldDef.Type.STRING
	model	FieldDef.Type.STRING
	class	FieldDef.Type.STRING
	color	FieldDef.Type.STRING
	value	FieldDef.Type.STRING
	tax	FieldDef.Type.STRING
	paid	FieldDef.Type.STRING

Table 9-2 Primary Key Field Names

Primary Key Field Names			
zipcode	lastname	first name	ssn

Table 9-3 Shard Key Field Names

Shard Key Field Names
zipcode

Upon examining this schema, one can see that the example `rmvTable` consists of rows of data the Registry of Motor Vehicles might maintain about vehicle owners who have registered a primary vehicle and (optionally) a second and maybe a third vehicle. In addition to personal information about each owner - such as name, address, gender, phone number(s), etc. - each row of data also contains an array in which each element of the array is a record whose contents consists of information about each vehicle the owner registers.

For example, in addition to vehicle attributes such as the make, model, color, etc., the record will also contain the vehicle's assessed value, registration fee (the tax), and whether or not the owner has paid the fee. Although the table schema presented above may seem a bit contrived, it is intended to demonstrate a broad spectrum of data types from the Oracle NoSQL Database table API.

To create a table with the above schema, the preferred approach is to employ the table Data Definition Language (DDL) (see Table Data Definition Language Overview), rather than entering individual commands in the store's admin CLI. To accomplish this, you can follow the instructions presented in the following sections to compile and

execute the LoadRmvTable program, which will populate the desired table after using the DDL to create it.

Create and Populate rmvTable with Example Data

Assuming an Oracle NoSQL Database store (either non-secure or secure) has been deployed with KVHOME equal to /opt/oracle/kv-ee, the LoadRmvTable program supplied in the separate Oracle NoSQL Database example distribution can be executed to create and populate the table named rmvTable. Before executing LoadRmvTable though, that program must first be compiled. To do this, assuming you have installed the example distribution under the directory /opt/oracle/nosql/apps/kv/examples, type the following from your client node's command line:

```
cd /opt/oracle/nosql/apps/kv
javac -classpath \
    /opt/oracle/kv-ee/lib/kvclient.jar:examples \
    examples/hadoop/hive/table/LoadRmvTable.java
```

This should produce the file:

```
/opt/oracle/nosql/apps/kv/examples/hadoop/hive/table/LoadRmvTable.class
```

How to Run LoadRmvTable When the Store is Non-Secure

To execute LoadRmvTable to create and populate the table named rmvTable with example data in a store configured for non-secure access, type the following at the command line of the client node, which must have network connectivity with a node running the admin service of the non-secure store you deployed (for example, kv-host-1 itself):

```
cd /opt/oracle/nosql/apps/kv
java -classpath \
    /opt/oracle/kv-ee/lib/kvclient.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.hive.table.LoadRmvTable \
    -store example-store -host kv-host-1 -port 5000 \
    -nops 79
    [-delete]
```

The following parameters are required: -store, -host, -port, and -nops, whereas the -delete parameter is optional.

In the example command line above, the argument -nops 79 requests that 79 rows be written to rmvTable. If more or less than that number of rows is desired, then the value of the -nops parameter should be changed.

If LoadRmvTable is executed a second time and the optional -delete parameter is specified, then all rows added by any previous executions of LoadRmvTable are deleted from the table prior to adding the requested new rows. Otherwise, all pre-existing rows are left in place, and the number of rows in the table will be increased by the requested -nops number of new rows.

How to Run LoadRmvTable When the Store is Secure

To execute `LoadRmvTable` against the secure store that you deployed and provisioned with a non-administrative user according to the steps presented in the Deploying a Secure Store appendix, an additional parameter must be added to the command line above. In this case, type the following on the command line:

```
java -classpath \
    /opt/oracle/kv-ee/lib/kvclient.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    hadoop.hive.table.LoadRmvTable \
    -store example-store -host kv-host-1 -port 5000 \
    -nops 79 \
    -security /tmp/example-user-client-pwdfile.login \
    [-delete]
```

As explained in the `CountTableRows` Support Programs appendix, the additional `-security` parameter in the command above specifies the location of the login properties file for the given user or alias. All other parameters are the same as for the non-secure case.

Summary

At this point, a table named `rmvTable`, populated with the desired example data , should exist in the Oracle NoSQL Database store you deployed. The data in that table can then be queried using the Hive Query Language (HQL).

Creating and Populating the exampleJsonTable

Similar to the programs that create and populate the example `vehicleTable` and `rmvTable`, the Oracle NoSQL Database example distribution provides another program, named `LoadJsonExample`, that you can use to create and populate a table consisting of rows with a field containing a JSON document. You can use the table created by the `LoadJsonExample` program to explore employing Hive to query JSON documents stored in an Oracle NoSQL Database.

This section describes the `LoadJsonExample` program, along with the schema employed when creating the desired example table, as well as how to compile and execute that program.

Schema for the Example Table Named exampleJsonTable

To demonstrate how a Hive query can be applied to an Oracle NoSQL Database table consisting of rows containing JSON documents, a table named `exampleJsonTable` having the schema shown in the table below is created in the Oracle NoSQL Database store deployed for this example. The data types specified in the schema shown below are defined by the Oracle NoSQL Database table API (see FieldDef.Type)

Table 9-4 Schema for exampleJsonTable

Field Name	Field Type
id	FieldDef.Type.INTEGER
jsonField	FieldDef.Type.JSON

Table 9-5 Primary Key Field Names

Primary Key Field Names
id

The `exampleJsonTable` will consist of rows with only two fields (columns). The first field contains a unique identification number that will be used as the primary key for the table. The rows of the second field will contain strings in valid JSON format; that is, a JSON document. The attributes of each JSON document in a given row specify information about current and former members of the United States senate; for example, the given senator's name, party affiliation, contact information, etc.

To create a table with the above schema, and populate that table with the desired JSON documents, follow the instructions presented in the next sections to compile and execute the `LoadExampleJson` program.

Create and Populate exampleJsonTable with Example Data

Assuming an Oracle NoSQL Database store (either non-secure or secure) has been deployed with KVHOME equal to `/opt/oracle/kv-ee`, the `LoadExampleJson` program supplied in the Oracle NoSQL Database example distribution can be executed to create and populate the table named `exampleJsonTable`. Before executing `LoadJsonExample` though, that program must first be compiled. To do this, assuming you have installed the example distribution under the directory `/opt/oracle/nosql/apps/kv/examples`, type the following from your client node's command line:

```
cd /opt/oracle/nosql/apps/kv
javac -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:examples \
    examples/hadoop/hive/es/table/LoadExampleJson.java
```

This should produce the file:

```
/opt/oracle/nosql/apps/kv/examples/  \
    hadoop/hive/es/table/LoadExampleJson.class
```

How to Run LoadJsonExample When the Store is Non-Secure

To execute `LoadExampleJson` when the store to contact is configured for non-secure access, type the following at the command line of the client node, which must have

network connectivity with a node running the admin service of the non-secure store you deployed (for example, `kv-host-1` itself):

```
cd /opt/oracle/nosql/apps/kv
java -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    [-Djava.util.logging.config.file=/opt/oracle/nosql/\
        apps/kv/examples/hadoop/hive/es/\
        example-logging.properties] \
    hadoop.hive.ex.table.LoadExampleJson \
    -store example-store -host kv-host-1 -port 5000 \
    -file /opt/oracle/nosql/apps/kv/examples/\
        hadoop/hive/ex/govtrack-senators-example.json \
    –table exampleJsonTable
    [-delete]
```

The following parameters are required: `-store`, `-host`, `-port`, `-file`, and `-table`, whereas the `-delete` parameter is optional.

In the example command line above, the argument `-file <filename>` requests that the contents of the specified file be retrieved and then written as JSON data to the `exampleJsonTable` that is created in the store. For convenience, the file specified above is provided with the example distribution. You can examine the contents of that file to see all of the possible document attributes that you can query.

If `LoadExampleJson` is executed a second time and the optional `-delete` parameter is specified, then all rows added by any previous executions of `LoadExampleJson` are deleted from the table prior to adding the requested new rows. Otherwise, all pre-existing rows are left in place, and duplicate rows (with new id values) will be added to the table.

How to Run LoadJsonExample When the Store is Secure

To execute `LoadExampleJson` against the secure store that you deployed and provisioned with a non-administrative user according to the steps presented in the Deploying a Secure Store appendix, an additional parameter must be added to the command line above. In this case, type the following on the command line:

```
java -classpath \
    /opt/oracle/kv-ee/lib/kvstore.jar:\
    /opt/oracle/kv-ee/lib/sklogger.jar:\
    /opt/oracle/kv-ee/lib/commonutil.jar:examples \
    [-Djava.util.logging.config.file=/opt/oracle/nosql/\
        apps/kv/examples/hadoop/hive/es/\
        example-logging.properties] \
    hadoop.hive.table.es.LoadExampleJson \
    -store example-store -host kv-host-1 -port 5000 \
    -file /opt/oracle/nosql/apps/kv/examples/\
        hadoop/hive/ex/govtrack-senators-example.json \
    –table exampleJsonTable \
    -security /tmp/example-user-client-pwdfile.login \
    [-delete]
```

As explained in the CountTableRows Support Programs appendix, the additional -security parameter in the command above specifies the location of the login properties file for the given user or alias. All other parameters are the same as for the non-secure case.

Summary

At this point, a table named exampleJsonTable, populated with the desired example data , should exist in the Oracle NoSQL Database store you deployed. The data in that table can then be queried using the Hive Query Language (HQL).

Configuring the Hive Client Environment

In order to use Apache Hive to query data in an Oracle NoSQL Database table, the Hive integration classes and other third party supporting classes provided by Oracle NoSQL Database must be made available to the Java VM of the Hive client, as well as the Java VMs of the data nodes making up the Hadoop cluster. This is accomplished by setting the value of the Hive client's HIVE_AUX_JARS_PATH environment variable to include each of the following JAR files provided with the Oracle NoSQL Database installation:

- kvclient.jar
- commonutil.jar
- sklogger.jar
- failureaccess.jar
- oraclepki.jar
- osdt_cert.jar
- osdt_core.jar
- antlr4-runtime-nosql-shaded.jar
- jackson-core.jar
- jackson-databind.jar
- jackson-annotations.jar

All JAR files specified by the HIVE_AUX_JARS_PATH environment variable will ultimately be added to the classpaths of the necessary VMs. Depending on the type of system, there are different options for adding the desired JAR files to your system's HIVE_AUX_JARS_PATH.

Copy Oracle NoSQL Database Libraries into Hive Auxiliary Directory

Some installations of Apache Hive provide a special directory to which third party libraries can be added so that direct modification of the HIVE_AUX_JARS_PATH envrionment variable is not necessary. Such installations employ facilities that automatically update the value of the HIVE_AUX_JARS_PATH with the JAR files located in that special directory.

For example, if your system is an Oracle Big Data Appliance (BDA) or an Oracle Big Data SQL system, then Hive, released by Cloudera, is installed as either packages or parcels (a binary distribution format that Cloudera provides as an alternative to

rpm/deb packages). Assuming your system's Hive installation is parcel based, then you would see a directory like the following:

```
/opt/cloudera/parcels/CDH/lib/hive/auxlib
```

For installations such as this, all JAR files located in the above directory will be automatcally added to the value of the HIVE_AUX_JARS_PATH environment variable whenever the Hive CLI is launched.

Thus, one way to make the necessary Oracle NoSQL Database JAR files available to the Hive classpath is to copy the necessary libraries into the auxiliary library provided by your Hive installation. For example, if your Hive installation provides a hive/auxlib directory like that shown above, you can do something like the following:

```
cd /opt/cloudera/parcels/CDH/lib/hive/auxlib

cp /opt/oracle/kv-ee/lib/kvclient.jar kvclient.jar
cp /opt/oracle/kv-ee/lib/commonutil.jar commonutil.jar
cp /opt/oracle/kv-ee/lib/sklogger.jar sklogger.jar
cp /opt/oracle/kv-ee/lib/failureaccess.jar \
    failureaccess.jar
cp /opt/oracle/kv-ee/lib/oraclepki.jar oraclepki.jar
cp /opt/oracle/kv-ee/lib/osdt_cert.jar osdt_cert.jar
cp /opt/oracle/kv-ee/lib/osdt_core.jar osdt_core.jar
cp /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar \
    antlr4-runtime-nosql-shaded.jar
cp /opt/oracle/kv-ee/lib/jackson-core.jar jackson-core.jar
cp /opt/oracle/kv-ee/lib/jackson-databind.jar \
    jackson-databind.jar
cp /opt/oracle/kv-ee/lib/jackson-annotations.jar \
    jackson-annotations.jar
```

To integrate Oracle NoSQL Database with Hive, it is important to copy the libraries shipped with Oracle NoSQL Database rather than linking to those libraries in the system's hive/auxlib directory. Copying the libraries shown above will prevent possible ClassLoader conflict errors that can be caused by older versions of third party libraries included in the system's Hadoop and Hive distributions.

Set HIVE_AUX_JARS_PATH in the Hive Client's hive-env.sh File

When you execute the hive command to enter the Hive CLI, the initialization script named hive-env.sh is executed; which sets the value of the HIVE_AUX_JARS_PATH environment variable for the Hive CLI. Rather than copying the Oracle NoSQL Database libraries in the manner described in the previous section, an alternative way to make the necessary Oracle NoSQL Database JAR files available to the Hive classpath would be to simply edit hive-env.sh and add those JAR files to the specification of the HIVE_AUX_JARS_PATH environment variable. For example,

```
edit <HIVE_CONF_DIR>/hive-env.sh

if [ -z "$HIVE_AUX_JARS_PATH" ];
then
export HIVE_AUX_JARS_PATH=\
```

```
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/oraclepki.jar,\
    /opt/oracle/kv-ee/lib/osdt_cert.jar,\
    /opt/oracle/kv-ee/lib/osdt_core.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/lib/\
        jackson-databind.jar,\
    /opt/oracle/kv-ee/lib/\
        jackson-annotations.jar
else
export HIVE_AUX_JARS_PATH=$HIVE_AUX_JARS_PATH,\
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/oraclepki.jar,\
    /opt/oracle/kv-ee/lib/osdt_cert.jar,\
    /opt/oracle/kv-ee/lib/osdt_core.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
    /opt/oracle/kv-ee/lib/jackson-core.jar,\
    /opt/oracle/kv-ee/lib/\
        jackson-databind.jar,\
    /opt/oracle/kv-ee/lib/\
        jackson-annotations.jar
```

> **Note:**
>
> Unlike setting a `CLASSPATH` environment variable, when setting the `HIVE_AUX_JARS_PATH` variable, the separator that is used is a comma, not a colon.

Set HIVE_AUX_JARS_PATH Directly on the Command Line

Instead of copying libraries, or editing the `hive-env.sh` script to make the necessary Oracle NoSQL Database JAR files available to the Hive classpath, you can always directly set the value of the `HIVE_AUX_JARS_PATH` environment variable on the command line before executing the Hive CLI. For example,

```
export HIVE_AUX_JARS_PATH=$HIVE_AUX_JARS_PATH \
    /opt/oracle/kv-ee/lib/kvclient.jar,\
    /opt/oracle/kv-ee/lib/commonutil.jar,\
    /opt/oracle/kv-ee/lib/sklogger.jar,\
    /opt/oracle/kv-ee/lib/failureaccess.jar,\
    /opt/oracle/kv-ee/lib/oraclepki.jar,\
    /opt/oracle/kv-ee/lib/osdt_cert.jar,\
    /opt/oracle/kv-ee/lib/osdt_core.jar,\
    /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar,\
```

```
/opt/oracle/kv-ee/lib/jackson-core.jar,\
/opt/oracle/kv-ee/lib/jackson-databind.jar,\
/opt/oracle/kv-ee/lib/\
    jackson-annotations.jar
```

Hive and Oracle NoSQL Database Security

With respect to running Hive queries against table data contained in a secure Oracle NoSQL Database store, a particularly important issue to address involves the creation and installation of an additional set of artifacts needed to communicate user credentials to the various components that participate in the execution of the Hive query. The additional artifacts that must be generated for Hive to work with a secure store are described in detail in the Model For Building & Packaging Secure Clients appendix, which describes the purpose of the artifacts, as well as how to generate and install them. Before proceeding, make sure you are familiar with the material presented in that section. Then employ the steps presented in the following sections to complete the configuration for running Hive queries against a secure store.

Generating the Login, Trust, and Password Artifacts

To execute a Hive query against a secure Oracle NoSQL Database store, the necessary public and private credentials must be incorporated in the definition of the Hive table that will be queried. To do this, in a fashion similar to that described in the Model For Building & Packaging Secure Clients appendix, you must create artifacts like those shown below, and store them on the Hive client's local system. For example, if you are using a password file to store the user's password, then you would generate files such as:

```
/tmp/kv-client-security
    client.trust
    hive-nosql.login
    example-user.passwd
```

Alternatively, if you are storing the user's password in an Oracle Wallet, then you would generate artifacts like the following:

```
/tmp/kv-client-security
    client.trust
    hive-nosql.login
    /example-user-wallet.dir
        cwallet.sso
```

Note that in both instances above, the file `hive-nosql.login` is identical to the file `example-user-server.login` that was generated in the Model For Building & Packaging Secure Clients appendix.

Generating the Server Side JAR File

After creating the login, trust, and password artifacts, you must generate a server side JAR file that can be added to Hive's `HIVE_AUX_JARS_PATH` environment variable.

Assuming you created artifacts like those described in the previous section, you would do the following:

```
cd /tmp/kv-client-security
jar cvf hive-nosql-server.jar client.trust
jar uvf hive-nosql-server.jar hive-nosql.login
```

The command above creates a JAR file named `hive-nosql-server.jar` with contents that include only public credentials which should look something like:

```
  0 META-INF/
 68 META-INF/MANIFEST.MF
508 client.trust
255 hive-nosql.login
```

Adding the Hive Client's Public Credentials to the Hive Environment

Afer creating the `hive-nosql-server.jar` file containing the Hive client's public credentials, the contents of that file must be made available to Hive via the `HIVE_AUX_JARS_PATH` environment variable. This is accomplished by following one of the options presented in the Configuring the Hive Client Environment appendix.

Summary

Once Hive has been configured for Oracle NoSQL Database security in the manner presented in this appendix, you can then incorporate the necessary artifacts in your Hive external table definition in the fashion presented previously in this document; for the `vehicleTable` from Primitive Data Types - The vehicleTable Example, and for the `rmvTable` from Non-Primitive Data Types - The rmvTable Example, and for the `exampleJsonTable` from NoSQL JSON Data Type - The exampleJsonTable Example.

Predicate Pushdown

To improve query performance, Apache Hive supports a mechanism referred to as predicate pushdown; in which the client side frontend processing of a given query decomposes the WHERE clause (the predicate) of the query into column information and corresponding comparison operators, passing (pushing) the resulting components to the database where search processing (filtering) can be performed on the database's server side backend. To achieve analogous performance improvements, the Oracle NoSQL Database table API Hive integration classes support similar predicate pushdown functionality when executing Hive or Big Data SQL queries against data in an Oracle NoSQL Database table. For example, consider the following query executed against the example `vehicleTable` whose schema was described previously in this document:

```
SELECT * FROM vehicleTable WHERE \
    type = 'auto' AND make = 'Chrysler' AND \
    model >= 'Imperial' AND model < 'Sebring';
```

This query will return all rows corresponding to automobiles (rather than trucks or SUVs) made by Chrysler; whose model is 'Imperial', 'Lebaron', or 'PTCruiser', but

not 'Sebring'. If predicate pushdown is not employed when executing this query, then all rows from the `vehicleTable` will be retrieved from the store's backend database and returned to the frontend client, where the predicate information will be applied to search the returned rows for the desired matches. On the other hand, if predicate pushdown is employed, then the information in the WHERE clause is sent to the store and all filtering is performed in the database itself, so that only the rows of the table that match the predicate are returned to the client. The predicate pushdown, when it can be employed, can result in significant performance improvements.

As the examples presented in this document demonstrate, the variety of predicates that can be employed when querying a table can be virtually unlimited. So it is important to understand that the predicates that can actually be pushed to the backend Oracle NoSQL Database store are restricted to a finite subset of all possible predicates. This is because the predicates that can be supported by Oracle NoSQL Database are not only dependent on what the Hive predicate pushdown mechanism supports, but the semantics of the Oracle NoSQL Database table API as well. As a result, the operators that are supported by the predicate pushdown mechanism of the table API Hive integration classes are currently limited to:

```
=   <   <=   >   >=   AND   OR   IN
```

In addition to the above set of operators, the semantics of the table API can also affect how the table's fields (columns) will be handled during predicate pushdown. Specifically, for a given query's predicate, if a valid primary key, index key, and/or field range (as defined by the table API) cannot be formed from all or a subset of that predicate's fields, and no part of the predicate can be pushed to the server using the filter mechanism provided by SQL for Oracle NoSQL Database (see Filtering Results in the SQL Beginner's Guide), then the query's predicate will not be decomposed and sent to the database for backend filtering. Instead, the system will fallback to the default mechanism, and perform all filtering on the client side, applying the predicate to all the rows in the given table.

For example, consider the query presented above. For that query, each component of the predicate satisfies the necessary criteria for pushdown, and so the whole predicate will be pushed to the database for search processing. To understand this, first observe that the operators referenced in the query's predicate belong to the set described above; that is, '=', 'AND', '>=', '<'.

Next, based on the schema of the `vehicleTable`, the fields named `type` and `make` form a valid primary key for performing a table scan; and the predicate components referencing the field named `model` form a valid field range. Compare this with a query such as,

```
SELECT * FROM vehicleTable WHERE make = 'Chrysler' AND \
    model >= 'Imperial' AND model < 'Sebring';
```

Assuming there is no index of the form (make, model), for this query, although the absence of the key's first field prevents the construction of a valid primary key as required by the semantics of the table API, the predicate can still be pushed to the backend store because it is considered valid for filtering by the SQL For Oracle NoSQL Database filtering mechanism. Finally, consider a query such as,

```
SELECT * FROM vehicleTable WHERE model LIKE "%Silverado%";
```

For this query, predicate pushdown will be bypassed and all filtering will be performed on the client side. This is because the predicate employs the `LIKE` operator, which is not currently eligible for predicate pushdown.

Note that the initial two example queries that were presented both result in the whole predicate being pushed and all filtering being performed on the backend. Whereas the third example query results in no predicate pushdown and all filtering being performed on the client side.

This does not mean that predicate pushdown will always be handled in such an all-or-nothing manner. On the contrary, for many queries, only part of the predicate will be pushed to the database to produce initial filtered results, which are then further filtered on the client side using the remaining - residual - part of the predicate.

For example, consider a query that wishes to find each '4WheelDrive' vehicle in the database that is 'blue', 'red', or 'yellow', and has a model name that begins with the letter 'E' (that is, Equinox, Expedition, Explorer, etc.). Such a query would look like the following:

```
SELECT * FROM vehicleTable WHERE \
    class = '4WheelDrive' AND \
    color IN ('blue','red','yellow') AND \
    model LIKE '%E%';
```

Based on the criteria presented in the next section, the only component of the query's predicate that cannot be pushed to the backend is the component that employs the `LIKE` operator (`model LIKE '%E%'`), whereas all other components in the query are eligible to be pushed. Thus, when executing the given query, the part of the predicate consisting of the components "`class = '4WheelDrive' AND color IN ('blue', 'red', 'yellow')`" will be pushed to the backend, producing rows referencing all four wheel drive vehicles that are blue, red, or yellow; after which the client will apply the residual predicate (`model LIKE '%E%'`) to the results from the backend, to select and return only those rows with model name including an uppercase 'E'.

Predicate Pushdown Criteria

When processing a given query that includes a predicate, the mechanism provided by the table API Hive integration classes will analyze the query's predicate and apply the following criteria to determine whether all, part, or none of the predicate can be pushed to the database for filtering on the backend.

- If the query's predicate includes components (column, operator, value) with comparison operators from the set {=, >, >=, <, <=}, as well as zero or more combinations of the AND conjunction, the OR conjunction, and/or one or more IN lists, then the predicate is eligible for predicate pushdown.
- Each combination of predicate components that form a valid (as defined by the table API) primary key, index key, or field range is eligible for predicate pushdown; using mechanisms that optimize for scale.
- If the query's predicate is found to contain multiple combinations that are eligible for predicate pushdown, then the combination resulting in optimal performance and scale will be pushed to the server. If more than one of those combinations is found to be optimal, then the first such combination will be pushed.
- Each component of the query's original predicate that is not included in the predicate to push will be added to the residual predicate for client side filtering.

- If all of the predicate components are found to be ineligible for predicate pushdown, then predicate pushdown will not be performed, and the system will fallback to the default mechanism, using the original predicate to perform only client side filtering.

It is important to understand the criteria listed above in conjunction with the data model and search patterns you expect to employ when you define the primary key, (along with any indexes), for a given Oracle NoSQL Database table that will be queried. Although the predicate pushdown mechanism will be employed automatically - without user intervention or special configuration - how you define your table and indexes can affect how well the more common queries you execute will perform and scale.

Predicate pushdown is employed automatically with no obvious indication (other than improved performance) that it is "on and working". As a result, if you wish to verify that the mechanism is indeed operating as described above, you can set the level of the following Oracle NoSQL Database loggers to the DEBUG level:

- `oracle.kv.hadoop.hive.table.TableStorageHandlerBase`
- `oracle.kv.hadoop.hive.table.TableHiveInputFormat`

After setting the level of the above loggers to DEBUG, you can run a query and then observe how the predicate pushdown mechanism processes the query's predicate by analyzing the contents of the logger output.

Part III

Integration with Oracle Big Data SQL

Topics

10

Introduction to Integration with Oracle Big Data SQL

This section describes the integration of Oracle NoSQL Database with Oracle Big Data SQL version 4.x. The goal of the Oracle Big Data SQL product is to allow users to employ the power of the Oracle SQL `SELECT` statement to manage and manipulate data stored in a number of different locations. Specifically, Oracle Big Data SQL is designed to provide SQL access to data stored in Apache Hadoop Distributed File System (HDFS), Apache Hive, various NoSQL databases - including Oracle NoSQL Database - as well as various relational databases. Oracle Big Data SQL achieves this by presenting Hadoop HDFS, Apache Hive, Oracle NoSQL Database, and the various other data sources as enhanced Oracle external tables of the Oracle Relational Database Management System (RDBMS) (See Managing External Tables in the *Oracle Database Administrator's Guide*). Oracle Big Data SQL maps the external semantics of accessing data from those sources - horizontal parallelism, location, and schema - to the Oracle Relational Database Management System's internal semantics. For more information on creating external table for Oracle NoSQL Database in Oracle Big Data SQL, see Create an External Table for Oracle NoSQL Database section in the *Oracle Big Data SQL User's Guide*.

To use Oracle Big Data SQL `SELECT` statements to query data stored in an Oracle NoSQL Database table, an Oracle Big Data SQL enabled external table must be created over the Oracle NoSQL Database table via an Apache Hive external table. In addition to presenting the steps to take to create such external tables, this document also presents a number of Oracle Big Data SQL queries that can be run against example table data contained in an Oracle NoSQL Database store, where the store can be either secure or non-secure.

Prerequisites

Before attempting to execute the examples that demonstrate the concepts presented in this section, you should first satisfy all prerequisites listed in both the Integration with Apache Hadoop MapReduce and the Integration with Apache Hive sections.

A Brief Primer on Oracle Big Data SQL

As stated in the Introduction to Integration with Oracle Big Data SQL, Oracle Big Data SQL allows SQL access to various external data sources such as an Oracle NoSQL Database table by presenting the data source as an Oracle external table. To achieve this, a mechanism referred to as an access driver is employed to access data as if it were a table in the Oracle relational database running in the Oracle Big Data SQL system. Oracle Big Data SQL extends the access driver mechanism of external tables by specifying a new access driver type for each data source that will be accessed. Prior to the introduction of Oracle Big Data SQL, the Oracle Database external tables mechanism defined only two access driver types:

- The `ORACLE_LOADER` access driver, for reading from flat files.

- The `ORACLE_DATAPUMP` access driver, for migrating data between Oracle databases in a proprietary format.

With the introduction of Big Data SQL, the following new access driver types are defined:

- The `ORACLE_HDFS` access driver, for accessing data stored in the Apache Hadoop Distributed File System.
- The `ORACLE_HIVE` access driver, for accessing data stored in Apache Hive tables or Oracle NoSQL Database tables.
- The `ORACLE_BIGDATA` access driver, for accessing files stored in an object store.

Both the `ORACLE_HDFS` and `ORACLE_HIVE` access drivers require the specification of a number of classes that satisfy the Apache Hadoop MapReduce programming model. Some of those classes are required by both access driver types, whereas some are required by only the `ORACLE_HIVE` access driver. The class types required by both `ORACLE_HDFS` and `ORACLE_HIVE` are:

- An instance of `org.apache.hadoop.mapreduce.InputFormat`
- An instance of `org.apache.hadoop.mapreduce.OutputFormat`
- An instance of `org.apache.hadoop.mapreduce.RecordReader`
- An instance of `org.apache.hadoop.mapreduce.InputSplit`

See package org.apache.hadoop.mapreduce.

The class types required by `ORACLE_HIVE` but not `ORACLE_HDFS` are:

- An instance of `org.apache.hadoop.hive.ql.metadata.HiveStoarageHandler` ; for example, the Oracle NoSQL Database `TableStorageHandler`.
- An instance of `org.apache.hadoop.hive.serde2.AbstractSerDe` ; for example, the Oracle NoSQL Database `TableSerDe`.
- An instance of `org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector` ; for example, the various `ObjectInspector` implementations defined by Oracle NoSQL Database and described in the Integration with Apache Hive section.
- An instance of `org.apache.hadoop.hadoop.hive.ql.metadata.HiveStoragePredicateHandler` .

See Hive API and package oracle.kv.hadoop.hive.table.

The `ORACLE_HDFS` access driver can only read data stored in HDFS files, whereas the `ORACLE_HIVE` access driver can read data stored not only in HDFS files, but data stored in other locations as well; for example an Oracle NoSQL Database table. As explained in the following sections, the integration of Oracle NoSQL Database with Apache Hive plays a prominent role in the integration of Oracle NoSQL Database with Oracle Big Data SQL.

11

Mapping the Oracle RDBMS Data Model to the Oracle NoSQL Database Table Model

As the examples in this section demonstrate, in order to execute an Oracle Big Data SQL query against data stored in an Oracle NoSQL Database table, a Hive external table must first be created with a schema mapped from the schema of the desired Oracle NoSQL Database table. Once that Hive external table is created, a corresponding Oracle RDBMS external table must then be created with a schema mapped from the schema of the Hive table. This is accomplished by applying the mappings shown in the following table:

Table 11-1 Data Type Mappings: Oracle NoSQL Database - Hive - RDBMS

Oracle NoSQL Database Type	Hive Type	RDBMS Type
FieldDef.Type.STRING	STRING	VARCHAR2(N)
	CHAR	
	VARCHAR	
FieldDef.Type.JSON	STRING	VARCHAR2(N)
FieldDef.Type.BOOLEAN	BOOLEAN	VARCHAR2(5)
FieldDef.Type.BINARY	BINARY	VARCHAR2(N)
FieldDef.Type.FIXED_BINARY	BINARY	VARCHAR2(N)
	TINYINT	
	SMALLINT	
FieldDef.Type.INTEGER	INT	NUMBER
FieldDef.Type.LONG	BIGINT	NUMBER
FieldDef.Type.FLOAT	FLOAT	NUMBER
FieldDef.Type.NUMBER	DECIMAL	NUMBER
FieldDef.Type.DOUBLE	DOUBLE	NUMBER
FieldDef.Type.ENUM	STRING	VARCHAR2(N)
FieldDef.Type.TIMESTAMP	java.sql.TIMESTAMP	TIMESTAMP
	DATE	
FieldDef.Type.ARRAY	ARRAY	VARCHAR2(N)
FieldDef.Type.MAP	MAP<STRING, data_type>	VARCHAR2(N)
FieldDef.Type.RECORD	STRUCT <col_name : data_type,…>	VARCHAR2(N)
	UNIONTYPE <data_type, data_type, …>	

It is important to understand that when using Oracle Big Data SQL to query data in an Oracle NoSQL Database table, the schema of the Oracle external table you create is dependent on the schema of the corresponding Hive external table; which, in turn, is dependent on the schema of the Oracle NoSQL Database table you wish to query. Thus, if either type of external table is created using a schema that includes a data type that does not belong to one of the mappings presented in the table above, then an error will occur when any attempt is made to query the table.

Note that for fields in the Oracle external table specified as VARCHAR2(N), the value of N is the maximum number of characters of the variable length STRING that represents the specified field in the corresponding Hive and Oracle NoSQL Database tables. Therefore, you should use the type, structure, and expected length or size of the corresponding Hive and Oracle NoSQL Database fields to determine the appropriate value to specify for `N` when creating the Oracle external table.

12

Executing SQL Queries Against Oracle NoSQL Database

The examples presented in this document were run using Oracle Big Data SQL 4.x and Oracle NoSQL Database 20.x, both installed on an Oracle Big Data Appliance (BDA). The system's Hive client environment was configured according to the directions presented in the Configuring the Hive Client Environment appendix.

Prior to attempting the examples presented here, first make sure you have satisfied all prerequisites described at the beginning of this document; which includes deploying an Oracle NoSQL Database store, either secure or non-secure.

Once the necessary prerequisites have been satisfied, follow the directions presented in the `CountTableRows` Support Programs appendix to create and populate the table named `vehicleTable` in the Oracle NoSQL Database. Then follow the directions presented in the Creating and Populating the rmvTable appendix and the Creating and Populating the exampleJsonTable appendix to create and populate the tables named `rmvTable` and `exampleJsonTable` respectively.

Finally, after satisfying all prerequisites and creating and populating each example table in the Oracle NoSQL Database store, follow the directions presented in the Configuring Oracle Big Data SQL For Querying Oracle NoSQL Database appendix to configure the Oracle Big Data SQL system for executing SQL queries against data stored in the Oracle NoSQL Database you deployed.

Mapping Hive External Tables to Oracle NoSQL Database Tables

In order to use Oracle Big Data SQL to query table data in an Oracle NoSQL Database store, you must first create and map a Hive external table to the desired table defined in the store. As described below, when the store is configured for security, the command to do this requires a few additional parameters.

Mapping Hive Tables to Oracle NoSQL Database Tables In a Non-Secure Store

Assuming you have deployed a non-secure Oracle NoSQL Database store in the manner described in the Deploying a Non-Secure Store appendix, login to one of the nodes of the Big Data SQL system that can be used as a Hive client. Then, from the Hive command line interface, execute the following command to map a Hive external table to the `vehicleTable` described in the `CountTableRows` Support Programs appendix, where line breaks are inserted for readability:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTable
    (type STRING, make STRING, model STRING, class STRING, color
STRING,
```

```
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "vehicleTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

Similarly, to map a Hive external table to the `rmvTable` described in the Creating and Populating the rmvTable appendix:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTable
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT street:STRING,
            unit:INT, city:STRING, zip:INT>,
        VEHICLEINFO ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "rmvTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

Finally, to map a Hive external table to the `exampleJsonTable` described in Creating and Populating the exampleJsonTable appendix:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleJsonTable
    (id INT, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3");
```

Mapping Hive Tables to Oracle NoSQL Database Tables In a Secure Store

Assuming you have deployed a secure Oracle NoSQL Database store in the manner described in the Deploying a Secure Store appendix, login to one of the nodes of the Big Data SQL system that can be used as a Hive client. Then, from the Hive command

line interface, execute the following command to use a password file to map a Hive external table to the `vehicleTable` in the secure Oracle NoSQL Database:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTablePasswd
    (type STRING, make STRING, model STRING, class STRING, color
STRING,
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "vehicleTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

And to use an Oracle Wallet to map a Hive external table to that same `vehicleTable` in the secure Oracle NoSQL Database, execute the Hive command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS vehicleTableWallet
    (type STRING, make STRING, model STRING, class STRING, color
STRING,
        price DOUBLE, count INT, dealerid DECIMAL, delivered TIMESTAMP)
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "vehicleTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.wallet.dir" = "/tmp/example-user-wallet.dir");
```

Similarly, to use a password file to map a Hive external table to the `rmvTable` described in Creating and Populating the rmvTable appendix, execute the command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTablePasswd
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT street:STRING,
        unit:INT, city:STRING, zip:INT>,
        vehicleinfo ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
```

```
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "rmvTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

And to use an Oracle Wallet to map a Hive external table to that same `rmvTable` in the secure Oracle NoSQL Database, execute the command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS rmvTableWallet
    (zipcode STRING, lastname STRING, firstname STRING, ssn BIGINT,
        gender STRING, license BINARY, phoneinfo MAP<STRING, STRING>,
        address STRUCT<number:INT street:STRING,
            unit:INT, city:STRING, zip:INT>,
        vehicleinfo ARRAY<STRUCT<type:STRING, make:STRING,
            model:STRING, class:STRING, color:STRING,
            value:FLOAT, tax:DOUBLE, paid:BOOLEAN>>)
    COMMENT 'Hive mapped to NoSQL table: rmvTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "rmvTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.wallet.dir" = "/tmp/example-user-wallet.dir");
```

Finally, to use a password file to map a Hive external table to the `exampleJsonTable` described in the Creating and Populating the exampleJsonTable appendix, execute the command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleTablePasswd
    (id INTEGER, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.pwdfile.file" = "/tmp/example-user.passwd");
```

And to use an Oracle Wallet to map a Hive external table to that same `exampleJsonTable` in the secure Oracle NoSQL Database, execute the command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS exampleJsonTableWallet
    (int INTEGER, jsonfield STRING)
    COMMENT 'Hive mapped to NoSQL table: exampleJsonTable'
    STORED BY 'oracle.kv.hadoop.hive.table.TableStorageHandler'
    TBLPROPERTIES (
        "oracle.kv.kvstore" = "example-store",
        "oracle.kv.hosts"= "kv-host-1:5000,kv-host-2:5000,kv-
host-3:5000",
        "oracle.kv.tableName" = "exampleJsonTable",
        "oracle.kv.hadoop.hosts" = "dn-host-1,dn-host-2,dn-host-3",
        "oracle.kv.security" = "/tmp/hive-nosql.login",
        "oracle.kv.ssl.truststore" = "/tmp/client.trust",
        "oracle.kv.auth.username" = "example-user",
        "oracle.kv.auth.wallet.dir" = "/tmp/example-user-wallet.dir");
```

Note that although the tables in the secure Oracle NoSQL Database store are named `vehicleTable`, `rmvTable`, and `exampleJsonTable`, the names of the tables created in Hive are not required to match the names of the corresponding Oracle NoSQL Database tables. This allows you to create different Hive tables mapped to the same Oracle NoSQL Database table; which allows you to query the Oracle NoSQL Database table using different security mechanisms.

Mapping Oracle RDBMS External Tables to Hive External Tables

At this point, although Hive queries can be executed against the table data in the Oracle NoSQL Database you deployed, you cannot yet execute SQL queries against that data. In order to use Oracle Big Data SQL to query that data, you must apply the data model mapping presented in the Table 11-1 table, along with the schemas defined for the Oracle NoSQL Database `vehicleTable`, `rmvTable`, and `exampleJsonTable` to create and map the corresponding Oracle Database external tables to each of the Hive tables that you created.

Assuming the Oracle Database has a pluggable database named `ORCLPDB1` and a user named `NOSQL_EXAMPLE_USER` with password `welcome1`, connect to the database via Oracle sqlplus and execute the SQL commands presented in the following sections to create the necessary Oracle external tables; for example,

```
sqlplus NOSQL_EXAMPLE_USER/welcome1@oracledb-host:1521/ORCLPDB1
```

Mapping Oracle RDBMS Tables to Hive Tables for Non-Secure Store

To create and map an Oracle Database external table to the Hive external table initially mapped to the Oracle NoSQL Database `vehicleTable`, execute commands like the following from the sqlplus prompt:

```
CREATE TABLE IF NOT EXISTS vehicleTable
    (type VARCHAR2(10), make VARCHAR2(12), model VARCHAR2(20),
```

```
        class VARCHAR2(40), color VARCHAR2(20), price NUMBER(8,2),
        count NUMBER, dealerid NUMBER, delivered TIMESTAMP)
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

Similarly, to map an Oracle Database external table to the Hive external table initially mapped to the Oracle NoSQL Database `rmvTable`, execute the command,

```
CREATE TABLE IF NOT EXISTS rmvTable
    (zipcode VARCHAR2(7), lastname VARCHAR2(20), firstname
VARCHAR2(20),
        ssn NUMBER, gender VARCHAR2(6), license VARCHAR2(9),
        phoneinfo VARCHAR2(67), address VARCHAR2(100),
        vehicleinfo VARCHAR2(1000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

Finally, to map an Oracle Database external table to the Hive external table initially mapped to the Oracle NoSQL Database `exampleJsonTable`, execute the command,

```
CREATE TABLE IF NOT EXISTS exampleJsonTable
    (id INT, jsonfield VARCHAR2(2000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

Note that if you want the name that you specify for the Oracle Database external table to be different than the name of the Hive external table to which it is mapped, then you must use the `com.oracle.bigdata.tablename` property to specify the name of the Hive external table in the command's `ACCESS PARAMETERS`; otherwise the name of the Oracle external table will default to the name of the Hive table. For example,

```
CREATE TABLE IF NOT EXISTS oracleVehicleTable
    (type VARCHAR2(10), make VARCHAR2(12), model VARCHAR2(20),
        class VARCHAR2(40), color VARCHAR2(20), price NUMBER(8,2),
        count NUMBER, dealerid NUMBER, delivered TIMESTAMP)
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log
        com.oracle.bigdata.tablename=vehicleTable))
    REJECT LIMIT UNLIMITED;
```

The Oracle Big Data SQL 4 User's Guide provides information on the various `ACCESS PARAMETERS` that can be specified for the `ORACLE_HIVE` access driver.

Mapping Oracle RDBMS Tables to Hive Tables for Secure Store

To create an Oracle Database external table for querying the Oracle NoSQL Database `vehicleTable` when a secure store is accessed via a password file, execute a command like the following from the sqlplus prompt,

```
CREATE TABLE IF NOT EXISTS vehicleTablePasswd
    (type VARCHAR2(10), make VARCHAR2(12), model VARCHAR2(20),
        class VARCHAR2(40), color VARCHAR2(20), price NUMBER(8,2),
        count NUMBER, dealerid NUMBER, delivered TIMESTAMP)
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

And when the store is accessed using an Oracle wallet execute,

```
CREATE TABLE IF NOT EXISTS vehicleTableWallet
    (type VARCHAR2(10), make VARCHAR2(12), model VARCHAR2(20),
        class VARCHAR2(40), color VARCHAR2(20), price NUMBER(8,2),
        count NUMBER, dealerid NUMBER, delivered TIMESTAMP)
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

To create an Oracle Database external table for querying the Oracle NoSQL Database `rmvTable`, when a secure store is accessed via a password file, execute the command,

```
CREATE TABLE IF NOT EXISTS rmvTablePasswd
    (zipcode VARCHAR2(7), lastname VARCHAR2(20), firstname
VARCHAR2(20),
        ssn NUMBER, gender VARCHAR2(6), license VARCHAR2(9),
        phoneinfo VARCHAR2(67), address VARCHAR2(100),
        vehicleinfo VARCHAR2(1000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

And when the store is accessed using an Oracle wallet execute,

```
CREATE TABLE IF NOT EXISTS rmvTableWallet
    (zipcode VARCHAR2(7), lastname VARCHAR2(20), firstname
VARCHAR2(20),
        ssn NUMBER, gender VARCHAR2(6), license VARCHAR2(9),
        phoneinfo VARCHAR2(67), address VARCHAR2(100),
        vehicleinfo VARCHAR2(1000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
```

```
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

To create an Oracle Database external table for querying the Oracle NoSQL Database `exampleJsonTable`, when a secure store is accessed via a password file, execute the command,

```
CREATE TABLE IF NOT EXISTS exampleJsonTablePasswd
    (id INT, jsonfield VARCHAR2(2000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

And when the store is accessed using an Oracle wallet execute,

```
CREATE TABLE IF NOT EXISTS exampleJsonTableWallet
    (id INT, jsonfield VARCHAR2(2000))
    ORGANIZATION EXTERNAL (TYPE ORACLE_HIVE
        DEFAULT DIRECTORY DEFAULT_DIR
        ACCESS PARAMETERS (com.oracle.bigdata.log.qc=query.log))
    REJECT LIMIT UNLIMITED;
```

13

Example: SQL Queries On Oracle NoSQL Database Tables

After creating the Oracle Database external tables described in the previous sections, you can then execute SQL `SELECT` queries to retrieve the data stored in the Oracle NoSQL Database store. To execute the example queries presented in the following sections, connect to the database via Oracle sqlplus and set the linesize to aid readability; for example,

```
sqlplus NOSQL_EXAMPLE_USER/welcome1@<oracledbhost>:1521/ORCLPDB1

set linesize 250;
```

Before executing the examples presented in the following sections, you can verify that each Hive table you created is now accessible from the system's Oracle Database. To display information about those tables, execute the following commands from the SQL prompt:

```
col cluster_id format A20;
col database_name format A15;
col owner format A10;
col table_name format A20;
col partitioned format A15;

SELECT cluster_id, database_name, owner, table_name,
    partitioned FROM all_hive_tables;
```

After verifying that the Hive tables you created are accessible, you can then query the `all_hive_columns` view to verify the data mappings you specified.

```
col table_name format A20;
col column_name format A20;
col hive_column_type format A20;
col oracle_column_type format A20;

SELECT table_name, column_name, hive_column_type,
    oracle_column_type FROM all_hive_columns;
```

Example Queries on the vehicleTable

When using Oracle Big Data SQL to query data in the Oracle NoSQL Database `vehicleTable`, simply query the Oracle external table you mapped to that table using the SQL `SELECT` command. For example,

```
col type format A5;
col make format A8;
col model format A15;
col class format A25;
col color format A16;
col price format $99,999.90;
col count format 99999;
col dealerid format 0.99999999999999999999999999999999999990;
col delivered format A30;

set linesize 500;

SELECT * FROM vehicleTable;
SELECT count(*) FROM vehicleTable;
SELECT min(price) FROM vehicleTable;
SELECT min(dealerid) FROM vehicleTable;

SELECT * FROM vehicleTable WHERE make='GM';
SELECT * FROM vehicleTable WHERE model='Camaro';
SELECT * FROM vehicleTable WHERE model LIKE '%Silverado%';
SELECT * FROM vehicleTable WHERE color='yellow'
    AND type > 'auto' AND type < 'truck';
SELECT * FROM vehicleTable WHERE
    type > 'auto' AND type <= 'truck' AND make LIKE 'Ford';
SELECT * FROM vehicleTable WHERE dealerid > 0.7 AND dealerid < 0.75;
SELECT * FROM vehicleTable WHERE delivered
    BETWEEN '06-JUN-12 06:53:41.448643 AM' AND '05-SEP-15
03:40:22.057282 PM'
    ORDER BY delivered;
```

Note that if you created the Oracle Database external tables `vehicleTablePasswd` and/or `vehicleTableWallet` for the case where the Oracle NoSQL Database store is secure, then you would simply replace the name `vehicleTable` in the queries above with `vehicletablePasswd` and/or `vehicleTableWallet`.

Example Queries on the rmvTable

To use Oracle Big Data SQL to query data in the Oracle NoSQL Database `rmvTable`, you can similarly execute basic queries like the following on the Oracle Database

external table you mapped to that table in the Oracle NoSQL Database store. For example,

```
SELECT * FROM rmvTable;


SELECT lastname,firstname,gender,address FROM rmvTable;


SELECT min(ssn) FROM rmvTable;


SELECT count(*) FROM rmvTable;


SELECT firstname,lastname,phoneinfo FROM rmvTable;


SELECT vehicleinfo FROM rmvTable;
```

More Example Queries on the rmvTable

To achieve more complicated query functionality, you can employ either Oracle regular expression functions such as `REGEXP_LIKE` and `REGEXP_SUBSTR`, or Oracle JSON operators such as `JSON_QUERY` and `JSON_EXISTS` (or a combination).

Example Queries Using Oracle Regular Expression Functions

The example queries presented here demonstrate the use of Oracle regular expression functions to query the Oracle NoSQL Database `rmvTable`.

To display the firstname, lastname, address, and vehicleinfo array from each row of the Oracle NoSQL Database `rmvTable` in which the value of the lastname column begins with the letter 'H', execute the query,

```
SELECT firstname,lastname,address,vehicleinfo FROM rmvTable WHERE
    REGEXP_LIKE (lastname, '^[H].*');
```

Recall that the `vehicleinfo` field of the Oracle NoSQL Database `rmvTable` is an array of Oracle NoSQL Database RECORD types that are mapped to the Oracle Database STRING type in which each element of a given RECORD is represented as name-value pairs when mapped to the Oracle Database data model; for example, '"make":"Chrysler"', '"color":"red"', '"paid":true', etc.

Suppose then, that you wish to list the name and address of each person in the database whose first or "primary" vehicle matches certain criteria. Additionally, suppose that rather than listing each element in the vehicleinfo array, you instead whish to list only the vehicle information related to the vehicle that matched the desired criteria. That is, you do not wish to list information about any other vehicles associated with a given owner.

For example, you might want to list all owners whose primary vehicle is made by GM, or all owners who own a Camaro. Or maybe you want to list all owners who have not yet registered their primary vehicle. If we assume that information about each owner's

primary vehicle is stored in the first element of the vehicleinfo array, then the queries below use Oracle regular expression functions to match on the sort of criteria just described. Specifically,

To find all owners with a primary vehicle made by GM:

```
SELECT firstname,lastname,address,
    REGEXP_SUBSTR(
        VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,1)
    "Primary make: GM"
    FROM rmvTable WHERE
    REGEXP_LIKE (
        REGEXP_SUBSTR(
            VEHICLEINFO,'\{([[:alnum:]":,\.]+){1,3}\}',1,1),
                   '"make":"GM"');
```

To find all owners whose primary vehicle is a Camaro:

```
SELECT firstname,lastname,address,
    REGEXP_SUBSTR(
        VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,1)
    "Primary model: Camaro"
    FROM rmvTable WHERE
    REGEXP_LIKE (
        REGEXP_SUBSTR(
            VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,1),
                '"model":"Camaro"');
```

To find all owners whose primary vehicle has not been registered:

```
SELECT firstname,lastname,address,
    REGEXP_SUBSTR(
        VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,1)
    "Primary fee NOT paid"
    FROM rmvTable WHERE
    REGEXP_LIKE (
        REGEXP_SUBSTR(
            VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,1),
                '"paid":false');
```

To find all owners whose second vehicle is a truck:

```
SELECT firstname,lastname,address,
    REGEXP_SUBSTR(
        VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,2)
    "Second vehicle type: truck"
    FROM rmvTable WHERE
    REGEXP_LIKE (
        REGEXP_SUBSTR(
            VEHICLEINFO, '\{([[:alnum:]":,\.]+){1,3}\}',1,2),
                '"type":"truck"');
```

Example Queries Using Oracle JSON Operators

The example queries presented in this section demonstrate the use of Oracle JSON operators to execute queries similar to those presented in the previous section using Oracle regular expression functions.

```
SELECT  firstname, lastname,  j.address.street,  j.address.city,
    j.address.state, j.vehicleinfo.model FROM rmvTable j;


SELECT JSON_QUERY(vehicleinfo, '$[0]' WITH CONDITIONAL WRAPPER)
    FROM rmvTable;


SELECT firstname, lastname, address,
    JSON_QUERY(vehicleinfo, '$[0]' WITH CONDITIONAL WRAPPER)
    AS "Primary Vehicle is made by GM"
    FROM rmvTable WHERE
    JSON_QUERY(vehicleinfo, '$[0].make' WITH CONDITIONAL WRAPPER)
    LIKE '%GM%';


SELECT  firstname, lastname, address,
    JSON_QUERY(vehicleinfo, '$[0]' WITH CONDITIONAL WRAPPER)
    AS "Primary Vehicle is a Camaro"
    FROM rmvTable WHERE
    JSON_QUERY(vehicleinfo, '$[0].model' WITH CONDITIONAL WRAPPER)
    LIKE '%Camaro%';


SELECT  firstname, lastname, address,
    JSON_QUERY(vehicleinfo, '$[0].model' WITH CONDITIONAL WRAPPER)
    AS "Primary Vehicle Model",
    JSON_QUERY(vehicleinfo, '$[0].value' WITH CONDITIONAL WRAPPER)
    AS "Primary Vehicle Value",
    JSON_QUERY(vehicleinfo, '$[0].tax' WITH CONDITIONAL WRAPPER)
    AS "Tax Owed",
    JSON_QUERY(vehicleinfo, '$[0].paid' WITH CONDITIONAL WRAPPER)
    AS "Tax Paid"
    FROM rmvTable WHERE
    JSON_QUERY(vehicleinfo, '$[0].make' WITH CONDITIONAL WRAPPER)
    LIKE '%GM%';
```

Example Queries on the exampleJsonTable

To use Oracle Big Data SQL to query data in the Oracle NoSQL Database `exampleJsonTable`, you can execute queries like the following on the Oracle Database

external table you mapped to that table in the Oracle NoSQL Database store. For example,

```
set linesize 500;


col id format 9999;
col jsonfield format A1000;


SELECT * FROM exampleJsonTable WHERE ROWNUM <= 5;
```

The following queries use various combinations of JSON dot notation, the `JSON_VALUE` operator, and the `JSON_QUERY` operator to query and display only specific attributes of the JSON document in each row of the Oracle NoSQL Database table.

Query Using Only JSON Dot Notation

```
col personal format A15;
col party format A15;

SELECT id, j.jsonfield.personal.firstname,
    j.jsonfield.personal.lastname, j.jsonfield.party
    FROM exampleJsonTable j
    WHERE j.jsonfield.party = 'Independent'
    ORDER BY j.jsonfield.person.lastname;
```

Query Using JSON Dot Notation and the JSON_VALUE Operator

```
col firstname format A15;
col lastname format A15;
col homephone format A12;
col workphone format A12;

SELECT id,
    JSON_VALUE(j.jsonfield, '$.personal.firstname') firstname,
    JSON_VALUE(j.jsonfield, '$.personal.lastname') lastname,
    JSON_VALUE(j.jsonfield, '$.personal.party') party,
    JSON_VALUE(j.jsonfield, '$.personal.address.home.phone') homephone,
    JSON_VALUE(j.jsonfield, '$.personal.address.work.phone') workphone
    FROM exampleJsonTable j
    ORDER BY j.jsonfield.party;
```

Query Using JSON Dot Notation and the JSON_QUERY Operator

```
col committee format A25;
col caucus format A25;

SELECT id, j.jsonfield.personal.firstname,
    j.jsonfield.personal.lastname,
    JSON_QUERY(j.jsonfield, '$.duties.committee' PRETTY WITH WRAPPER)
committee,
    JSON_QUERY(j.jsonfield, '$.duties.caucus' PRETTY WITH WRAPPER)
```

```
caucus
    FROM exampleJsonTable j
    WHERE j.jsonfield.party = 'Democrat' AND ROWNUM <= 5;
```

Query Using JSON Dot Notation With Both JSON_VALUE and JSON_QUERY

```
col contrib format A12;
col committee format A50;
col contrib format A50;

SELECT
    JSON_VALUE(j.jsonfield, '$.personal.firstname') firstname,
    JSON_VALUE(j.jsonfield, '$.personal.lastname') lastname,
    j.jsonfield.contrib,
    j.jsonfield.party,
    JSON_QUERY(j.jsonfield, '$.duties.committee' PRETTY WITH WRAPPER)
committee,
    JSON_QUERY(j.jsonfield, '$.duties.caucus' PRETTY WITH WRAPPER)
caucus
    FROM exampleJsonTable j
    WHERE j.jsonfield.party = 'Republican'  AND ROWNUM <= 5;
```

14

Appendix

Topics

Configuring Oracle Big Data SQL For Querying Oracle NoSQL Database

In order to use Oracle Big Data SQL to query data in an Oracle NoSQL Database table, the Hive integration classes and other third party supporting classes provided by Oracle NoSQL Database must be made available to the Java VMs launched by Oracle Big Data SQL. This is accomplished by setting the value of the `java.classpath.oracle` system property to include each of the following JAR files provided with the Enterprise Edition of Oracle NoSQL Database:

- `kvclient.jar`
- `commonutil.jar`
- `sklogger.jar`
- `failureaccess.jar`
- `oraclepki.jar`
- `osdt_cert.jar`
- `osdt_core.jar`
- `antlr4-runtime-nosql-shaded.jar`
- `jackson-core.jar`
- `jackson-databind.jar`
- `jackson-annotations.jar`

In an Oracle Big Data SQL system, the value of the `java.classpath.oracle` system property is set in the configuration file named, `bigdata.properties`; which is located in a directory of the form,

```
/opt/oracle/product/18c/dbhome_1/bigdatasql/databases/ORCLCDB/
bigdata_config
```

The subdirectory `ORCLCDB` is the instance name of the Oracle 18c Database deployed to the Oracle Big Data SQL system on which the examples presented here were developed. You may need to adjust some of the path components for your particular environment.

Upon examining the contents of the `bigdata.properties` file, you should see an entry that looks like,

```
java.classpath.oracle=\
    /opt/oracle/product/18c/dbhome_1/bigdatasql/jlib/*:\
    /opt/oracle/product/18c/dbhome_1/jlib/orai18n.jar
```

Although you can explicitly add the necessary Oracle NoSQL Database libraries to the `java.classpath.oracle` system property by modifying the above entry in the `bigdata.properties` configuration file, the recommended way is to copy those libraries in the directory, `/opt/oracle/product/18c/dbhome_1/bigdatasql/jlib` ; that is,

```
cd /opt/oracle/product/18c/dbhome_1/bigdatasql/jlib

cp /opt/oracle/kv-ee/lib/kvclient.jar kvclient.jar
cp /opt/oracle/kv-ee/lib/commonutil.jar commonutil.jar
cp /opt/oracle/kv-ee/lib/sklogger.jar sklogger.jar
cp /opt/oracle/kv-ee/lib/failureaccess.jar failureaccess.jar
cp /opt/oracle/kv-ee/lib/oraclepki.jar oraclepki.jar
cp /opt/oracle/kv-ee/lib/osdt_cert.jar osdt_cert.jar
cp /opt/oracle/kv-ee/lib/osdt_core.jar osdt_core.jar
cp /opt/oracle/kv-ee/lib/antlr4-runtime-nosql-shaded.jar \
    antlr4-runtime-nosql-shaded.jar
cp /opt/oracle/kv-ee/lib/jackson-core.jar jackson-core.jar
cp /opt/oracle/kv-ee/lib/jackson-databind.jar jackson-databind.jar
cp /opt/oracle/kv-ee/lib/jackson-annotations.jar jackson-annotations.jar
```

To integrate Oracle NoSQL Database with the Big Data SQL query mechanism, it is important to copy the libraries shipped with Oracle NoSQL Database rather than linking to those libraries in the system's `bigdatasql/jlib` directory. Copying the libraries shown above will prevent possible `ClassLoader` conflict errors that can be caused by older versions of third party libraries included in the system's classpath.

Note that copying the libraries in the manner shown above is required for executing Big Data SQL queries against data in an Oracle NoSQL Database. But if you will also be executing Hive queries from one of the Big Data SQL system's database nodes, then in addition to copying the Oracle NoSQL Database libraries into `/opt/oracle/product/18c/dbhome_1/bigdatasql/jlib`, you must also copy those same libraries into the following directories on the database node from which the Hive queries will be executed:

```
    /opt/oracle/bigdatasql/bdcell-12.1/jlib-bds
and
    /opt/oracle/bigdatasql/bdcell-12.2/jlib-bds
```

Configuring Oracle Big Data SQL For Querying Data in a Secure Store

The Hive and Oracle NoSQL Database Security appendix describes the additional security artifacts that must be generated and installed to support executing queries (both Hive and Big Data SQL) against table data stored in a secure Oracle NoSQL Database. Those artifacts include the login, trust, and password artifacts, as well as the server side JAR file that contains the necessary public credentials for

communication with the secure store. After generating and installing the necessary security artifacts in the manner described in that section, add the server side JAR file to the Big Data SQL system's `java.classpath.oracle` system property.

For example, if the server side JAR file is named `hive-nosql-server.jar` and is installed in a directory such as `/tmp/kv-client-security`, then do the following:

```
cd /opt/oracle/product/18c/dbhome_1/bigdatasql/jlib
ln –s /tmp/kv-client-security/hive-nosql-server.jar \
    hive-nosql-server.jar
```

Additionally, if you will also be executing Hive queries from any of the Big Data SQL system's database nodes, then you must create the same link as that shown in the following directories on each such node,

```
    /opt/oracle/bigdatasql/bdcell-12.1/jlib-bds
and
    /opt/oracle/bigdatasql/bdcell-12.2/jlib-bds
```

Part IV

Integration with Elastic Search for Full Text Search

Topics

15

About Full Text Search

Topics

About Full Text Search

Full Text Search provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query.

Full Text Search will find all documents containing given query terms and return them in order of their similarity to the query. Notions of query and similarity are very flexible and depend on the specific application. The simplest search considers query as a set of words and similarity as the frequency of query words in the document.

In concert with the table interface, Oracle NoSQL Database integrates with the Elasticsearch third-party open-source search engine to enable Full Text Search capability against data stored in an Oracle NoSQL Database table. See Elasticsearch.

Full Text Search is an important aspect of any big data or database system. Users expect that when they input text into a box and click **search**, they will get the relevant search results they are looking for. Thus, besides providing high performance Full Text Search of data stored in Oracle NoSQL tables, the mechanism described in this document also allows users to explore a collection of information by applying multiple Elastisearch filters.

The feature described here provides a mechanism for marking fields from an Oracle NoSQL Database table schema as being text searchable. This so-called Oracle NoSQL Text Indexing mechanism allows one to create Elastisearch indexes on the data stored in Oracle NoSQL Database tables. It does this by causing the data in the indexed fields to automatically be stored in a corresponding index created in a given Elastisearch cluster. Once the data is stored (indexed) in Elastisearch, one can then use any native Elastisearch API to search and retrieve the data that matches the specified search criteria. References contained in the documents returned by Elasticsearch can then be used to retrieve the original Oracle NoSQL Database records that correspond to the indexed data.

> **Note:**
>
> So that the maintenance of indexes does not affect the performance of an Oracle NoSQL Database store, the text indexes that are used for Full Text Search will not be maintained locally by Oracle NoSQL Database components. Rather, they will instead be maintained by a remote Elasticsearch service hosted on other nodes.

Prerequisite to Full Text Search

In order to employ the Full Text Search feature, you need a running Oracle NoSQL Database store, and an Elasticsearch cluster. The Elasticsearch cluster must be reachable over a network from the Oracle NoSQL Database store. For performance reasons, when running in a production environment, the nodes making up the Oracle NoSQL Database, as well as the nodes of the Elasticsearch cluster should be separate hosts in a distributed environment, communicating over a network.

Currently, the Full Text Search feature of Oracle NoSQL Database will work with Elasticsearch version 2 (example: 2.4.6), but not versions greater than or equal to version 5. The following references can help you download, install, and start a version of Elasticsearch compatible with Oracle NoSQL Database:

- https://www.elastic.co/downloads/past-releases/elasticsearch-2-4-6
- https://www.elastic.co/guide/en/elasticsearch/reference/2.4/index.html

Once your Elasticsearch cluster is running, it should consist of one or more nodes. Some or all of the nodes will have services listening on two ports:

- The HTTP port, which is used for REST requests (default `9200`).
- The Elasticsearch transport port, used for communication between Elasticsearch nodes (default `9300`).

> **Note:**
>
> As explained in the sections below, you must know the HTTP port and the host name of at least one node in the Elasticsearch cluster. You also must know the name of the cluster itself, which by default is `elasticsearch`. This information must be provided to the Oracle NoSQL Database store so that it can find and connect to the Elasticsearch cluster.

16

Intergrating Elasticsearch with Oracle NoSQL Database

Topics

Registering Elasticsearch with Oracle NoSQL Database

Before you can use Oracle NoSQL Database to create a Text Index in an Elasticsearch cluster, you must register the desired cluster with the Oracle NoSQL Database store, using the `plan` command named `register-es`. It is via the `register-es` plan that you provide the name of the Elasticsearch cluster, the name of one of the hosts in that cluster, and the HTTP port on which that host is listening for connection requests. Specifically, the `register-es` plan command takes the following form:

```
plan register-es
        –clustername <name>
        -host <host|ip>
        -port <http port>
        -secure <true|false>
        [-wait]
        [-force]
```

For example, if your Elasticsearch cluster is named `elasticsearch` (the default) and includes a node running on your local host, listening on the default HTTP port (`9200`), then you would execute the following command from the Oracle NoSQL Database administrative command line interface (Admin CLI):

```
kv-> plan register-es
        –clustername elasticsearch
        –host 127.0.0.1
        –port 9200
        –secure false
        -wait

Executed plan 22, waiting for completion...
Plan 22 ended successfully
```

> **Note:**
>
> When the `register-es` plan is executed, if the Elasticsearch cluster specified in the command already contains indexes created under a registration between a previous NoSQL store instance and the current Elasticsearch cluster, then the plan will fail and display an error message. Such indexes are referred to as **stale indexes**, and the plan fails in the face of such stale indexes because the indexes currently maintained in the Elasticsearch cluster's state are associated with the store that previously created them under the original registration. Although those existing indexes are part of the Elasticsearch cluster's state, they are not part of the state of the new store instance. Allowing the new store instance to create a new registration through which new indexes can be created in the cluster can produce inconsistencies and possible conflicts between the state maintained by the store and the state maintained by Elasticsearch; resulting in potential error conditions.
>
> To avoid such error conditions, when the new store instance receives a request for a new registration with the Elasticsearch cluster, and that cluster contains indexes associated with a registration created by a previous store instance, the request is rejected; unless the `force` flag is specified. If the `force` flag is specified in the `register-es` command, then the store will request that Elasticsearch first remove all of its stale indexes; and only after those indexes have been successfully removed, will the registration be created between the new store instance and the Elasticsearch cluster.
>
> During the registration process the store's Admin Service (or simply, the Admin) verifies the existence of (as well as a network path to) the Elasticsearch node specified in the `register-es` command arguments, and then acquires from that node a complete list of connection information for all the nodes making up that Elasticsearch cluster. This information is stored in the Admin's state, as well as distributed to all the nodes of the Oracle NoSQL Database store. If the Elasticsearch cluster's population of nodes changes significantly (due to node or network failure, cluster reconfiguration, and so on), the `register-es` command can be repeated to update the Oracle NoSQL Database's list of Elasticsearch node connections.

After successfully executing the `register-es` plan, you can verify that the Oracle NoSQL Database store is indeed registered with the desired Elasticsearch cluster by executing the `show parameters` command from the Admin CLI in the following way:

```
show parameters –service <storage node id>
```

The `show parameters` command displays a list of properties for the specified storage node that, if the registration was successful, will include the name of the Elasticsearch cluster, along with the host names and/or IP addresses of the nodes making up that cluster. When you execute the `show parameters` command, the

value of the properties named `searchClustername` and `searchClusterMembers` will provide that information for you. For example,

```
kv-> show parameters -service sn1

capacity=1
haHostname=localhost
haPortRange=5005,5007
hostname=localhost
memoryMB=0
mgmtClass=oracle.kv.impl.mgmt.NoOpAgent
mgmtPollPort=0

mgmtTrapPort=0

numCPUs=8

registryPort=5000

rnHeapMaxMB=0

rnHeapPercent=85

rootDirPath=./kvroot
searchClusterMembers=127.0.0.1:9200
searchClusterName=elasticsearch
serviceLogFileCount=20
serviceLogFileLimit=2000000
storageNodeId=1

systemPercent=10
```

Deregistering Elasticsearch from an Oracle NoSQL Store

Oracle NoSQL Database implements *'one store, one Elasticsearch cluster'* policy. That is, a given store cannot be simultaneously registered with more than one Elasticsearch cluster. This policy is expressed through the registration model. See Registering Elasticsearch with Oracle NoSQL Database. Thus, if your store is currently registered with one Elasticsearch cluster, but you wish to register with a second cluster, then you must first deactivate – or deregister – the current registration. This is accomplished by executing the following `deregister-es` plan.

```
plan deregister-es [-wait]
```

> **Note:**
>
> Because of the one store, one cluster policy, the `deregister-es` command takes no arguments.

The store cannot deactivate a registration unless all indexes created under that registration have first been deleted from the Elasticsearch cluster. This can be

accomplished by executing the `DROP INDEX` command on each of the Full Text Indexes created by the store and located in the Elasticsearch cluster with which the store is registered. That is, from the Admin CLI, a command with the following form should be executed for each index:

```
execute 'DROP INDEX [IF EXISTS] <index> ON <table>';
```

Since the Elasticsearch cluster is created, maintained, and administered separate from the Oracle NoSQL Database store, that cluster may contain indexes that were created outside of the store's control, using the Elasticsearch API. These sort of indexes are not known to the store (are not in the store's state), and do not need to be deleted from the cluster in order to deactivate the store's registration with the cluster. Only the indexes that were created in the Elasticsearch cluster via the Oracle NoSQL `CREATE FULLTEXT INDEX` command must be deleted for the `deregister-es` command to succeed.

If the `deregister-es` command fails because the cluster still contains Full Text Indexes created by the store, the output for the command will display the names of the indexes that must be dropped. For example,

```
kv-> plan deregister-es –wait
Cannot deregister ES because these text indexes exist:

mytestIndex
JokeIndex

kv-> execute 'DROP INDEX mytestIndex ON myTable';
Plan 16 completed successfully

kv-> execute 'DROP INDEX JokeIndex ON myTable';
Plan 17 completed successfully

kv-> plan deregister-es –wait
Plan 18 completed successfully
```

The show parameters command can then be executed again, and its output examined, to determine if the store is indeed no longer registered with the Elasticsearch cluster.

> **Note:**
>
> There are two index types that can exist in Oracle NoSQL Database: a regular or Secondary Index, and a Text Index (for Full Text Search). With respect to index creation or deletion, although separate statements are needed for index creation (to distinguish the type of index to create), the same `DROP INDEX` statement is used to remove either type of index. When applied to a text index, a `DROP INDEX` command like those shown above not only stops the population of the index from the associated Oracle NoSQL Database table, but also removes the mapping and all related documents from Elasticsearch.

17
Managing Full Text Index

Topics

Creating a Full Text Index

Review the concepts of Oracle NoSQL Database tables and indexes for better understanding of this section. See Indexes in the *SQL Reference Guide*. That chapter describes the main type of index you can create on the fields of a given Oracle NoSQL Database table.

This section introduces a second type of index that can be created on a given table. This second index category – separate from the Secondary Indexes described in the *SQL Reference Guide* – is referred to as a *Full Text Index* or, simply, a *Text Index* on the associated table.

As with any index, a Text Index as defined here, allows one to search for rows of an Oracle NoSQL Database table having fields that share some common value or characteristic. The difference between the two types of indexes is that an Oracle NoSQL Database Secondary Index is created, maintained and queried all within the Oracle NoSQL Database store; using the Oracle NoSQL Database Table API. On the other hand, the creation of a Text Index is only initiated via the Table API. Although the store maintains information about the Text Indexes that are created, such indexes are actually created, maintained, and queried in the Elasticsearch cluster with which the store is registered (using the Elasticsearch API).

It is important to understand that when the first type of index is created, data from the indexed fields of the associated table are written to the store itself; whereas when a Text Index is created, that data is streamed to the Elasticsearch cluster with which the store is registered, which stores (indexes) the data so that the Elasticsearch API can be used to execute full text searches against that data. Whenever new data is written to, or existing data is deleted from the table, the corresponding Text Index located in the cluster is updated accordingly.

To index one or more fields of an Oracle NoSQL table for Full Text Search in Elasticsearch, you can use the store's Admin CLI to execute a command with the following format:

```
execute 'CREATE FULLTEXT INDEX
  [IF NOT EXISTS]
  <index> ON <table>
  (<field> {<mapping-spec>},<field> {<mapping-spec>}, ...)
  [ES_SHARDS=<n>]
  [ES_REPLICAS=<n>]
  [OVERRIDE]
  [COMMENT "<comment>"]';
```

Each argument, flag, and directive is described as follows, where any item encapsulated by square brackets `[...]` is optional, and the items encapsulated by curly braces `{...}` are required only when the field's value is a JSON document, but is optional otherwise:

- `index` - The name of the Text Index to create.
- `table` - The name of the table containing the fields to index.
- `field` - A comma-separated list of each field to index, encapsulated by open parentheses.
- Each field to index can optionally be associated with a mapping specification that specifies how Elasticsearch should handle the corresponding field. For example, whether Elasticsearch should treat the field's value as a text, number, date type, and so on; as well as what analyzer should be employed when indexing a text value. As explained in the sections below, the mapping specification for a given field must be expressed in valid JSON format.
- If the command above is executed and a Text Index with the specified name already exists, then unless the optional directive `IF NOT EXISTS` is specified, or the optional directive `OVERRIDE` is specified, the command will fail, displaying an error message. Specifying `IF NOT EXISTS` when the named index already exists will result in a *no-op*. If `OVERRIDE` is specified for an existing index, then the existing index will be deleted from Elasticsearch and a new index will be created with the same name.
- If the optional `ES_SHARDS` argument is specified, along with a corresponding `integer` value, then the setting for the number of primary shards Elasticsearch will use for the new index will be changed to the given value. Note that the default value for this setting is `5`, and this setting cannot be changed after the index has been created.
- If the optional `ES_REPLICAS` argument is specified, along with a corresponding `integer` value, then the setting for the number of copies of the indexed value each primary shard should have will be changed to the given value. Note that the default value for this setting is `1`, and this setting can be changed on a *live* index at any time.

For more information on how the value of the `ES_SHARDS` and `ES_REPLICAS` properties are used, refer to the Elasticsearch settings named `number_of_shards` and `number_of_replicas` described in the Elasticsearch documentation. See Elasticsearch Index Settings.

When `CREATE FULLTEXT INDEX` executes successfully, the Text Index name provided in the command (along with metadata associated with that name) is stored and maintained in the Oracle NoSQL store. Additionally, a corresponding text searchable index – the index that is actually queried – is also created in the Elasticsearch cluster with which the store is registered. Whereas the name associated with the index in Oracle NoSQL is the simple index name specified in the `CREATE FULLTEXT INDEX` command, the name of the corresponding index in Elasticsearch takes the following dot-separated form:

```
ondb.<store>.<table>.<index>
```

Each of the coordinates of the Elasticsearch index name will always be lowercase; even if their counterpart in Oracle NoSQL was specified as mixed or upper case. The first coordinate (or prefix) of the name will always be `ondb`; which distinguishes the indexes in Elasticsearch that were created by the Oracle NoSQL `CREATE FULLTEXT INDEX` command from other indexes created externally, via the Elasticsearch API. The `store` coordinate of the Elasticsearch index name is the name of the Oracle NoSQL Database store that asked Elasticsearch to create the index. And the `table` and `index` coordinates are the values specified for the corresponding arguments in `CREATE FULLTEXT INDEX`; that is, the name of the Oracle NoSQL table from which the values to index are taken, and the name of the Oracle NoSQL Text Index the store should maintain. Using the coordinates of any such index name in Elasticsearch, one should always be able to determine the origin of the data stored in the index.

Once you have executed the `CREATE FULLTEXT INDEX` command described above, you can verify that the Text Index has been successfully created in Oracle NoSQL by executing the show indexes command from the Admin CLI; for example,

```
kv-> show indexes –table mytestTable

Indexes on table mytestTable mytestIndex (...), type: TEXT
```

You can also verify that the corresponding full text searchable index has been created in Elasticsearch. To do this you can execute a `curl` command from the command line of a host with network connectivity to one of the nodes in the Elasticsearch cluster; for example,

```
curl –X GET 'http://esHost:9200/_cat/indices'

yellow open ondb.kvstore._checkpoint ...
yellow open ondb.kvstore.mytesttable.mytestindex ...
```

Notice the entry that references the `ondb.kvstore._checkpoint` index. This index will be automatically created upon the creation of the first Oracle NoSQL Text Index. Unless it is manually deleted from the Elasticsearch cluster, it will always appear in the output of the indices command. This so-called `_checkpoint` index contains internal information written by Oracle NoSQL to support recovery operations when Oracle NoSQL is restarted. In general, this index should never be removed or modified.

> **Note:**
>
> Throughout this document, the `curl` utility program is used to demonstrate how to issue and display the results of HTTP requests to the Elasticsearch cluster. The `curl` program is supported on most operating systems (linux, Mac OS X, Microsoft Windows, and so on). It is used here because it is easy to install and can be run from the command line. Other options you can explore for sending queries to Elasticsearch are:
>
> - The *elasticsearch-head* tool; which is a web front end for browsing, querying, and interacting with an Elasticsearch cluster. See elasticsearch-head.
> - The Elasticsearch Java API; which can be used to query Elasticsearch from within program control. See Elasticsearch Java API.

In addition to executing show indexes from the Oracle NoSQL Admin CLI, you can also execute the show table command; which, in addition to the table structure, will also list all indexes (both secondary and text) created for that table. For example,

```
kv-> show table –table mytestTable

{
  "json_version" : 1,
  "type" : "table",
  "name" : "mytestTable",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [
  {
    "name" : "id",

    "type" : "INTEGER",

    "nullable" : false,

    "default" : null

},
  {

    "name" : "category",

    "type" : "STRING",

    "nullable" : true,

 "default" : null

  },
  {

    "name" : "txt",
```

```
  "type" : "STRING",

    "nullable" : true,

    "default" : null

  } ],

  "indexes" : [
    {

"name" : "mytestIndex",
      "table" : "mytestTable",
      "type" " "text",
      "fields" : [ "category", "txt" ],
      "annotations" : {
        "category" : "{\"type\" : \"string\",
                       \"analyzer\" : \"standard\"}",
        "txt" : "{\"type\" : \"string\",
                  \"analyzer\" : \"english\"}"
      }
    } ]
}
```

> **Note:**
>
> You cannot evolve a Text Index created in Elasticsearch via the `CREATE FULLTEXT INDEX` mechanism. If you want to change the index definition, for example, add more columns to the index, you must first delete the existing index using the `DROP INDEX` command and then use `CREATE FULLTEXT INDEX` to create a new Text Index satisfying the desired definition.

Mapping a Full Text Index Field to an Elasticsearch Field

Unlike the command used to create a secondary index on data stored in an Oracle NoSQL table, the `CREATE FULLTEXT INDEX` command allows you to specify finer control over how Elasticsearch treats the fields to be indexed. For each field that you want Elasticsearch to handle in a non-default fashion, you can specify how you want Elasticsearch to treat that field's values by including a mapping specification with each such field when executing the `CREATE FULLTEXT INDEX` command.

If no mapping specification is provided for a given field, and if that field contains any indexable Oracle NoSQL data type – except JSON data – then Oracle NoSQL will use that data type to determine the appropriate type with which to map the field's values to the Elasticsearch type system. This means that for fields containing non-JSON data, the mapping specification can be used to enforce and/or override the data type Elasticsearch should use when indexing the field's contents.

For example, if a field of a given table contains values stored as the Oracle NoSQL Database `string` type, then the default mapping supplied to Elasticsearch will declare

that values from that field should be indexed as the Elasticsearch `string` type. But if you want Elasticsearch to treat the values of that field as the Elasticsearch `integer` type, then you would provide a mapping specification for the field including an explicit type declaration; that is,

```
{"type":"integer"}
```

But care must be taken when mapping incompatible data types. For the example just described, Elasticsearch will encounter errors if any of the `string` values being indexed contain non-numeric characters. See Elasticsearch Mapping.

For the case where the field to be indexed has values that are JSON documents, a mapping specification must always be provided in the `CREATE FULLTEXT INDEX` command; otherwise an error will occur. A mapping specification is necessary for such fields because, as explained later, it is not the document itself that is indexed, but a subset of the document's fields. When a JSON document is stored in an Oracle NoSQL Database table, Oracle NoSQL knows only that a value of type JSON was stored. It does not know the type intended for any of the fields (attributes) within the document. Thus, for each of the document's fields that will be indexed, the user must provide a corresponding mapping specification that specifies the type that Elasticsearch should use when indexing the field's value.

In addition to specifying the data type of a given field's content, the mapping specification can also be used to further refine how Elasticsearch processes the data being indexed. This is accomplished by including an additional set of parameters in the mapping specification. For example, suppose you want Elasticsearch to apply an analyzer different than the default analyzer when indexing a field with content of type string. In this case, you would specify a mapping specification of the form:

```
{"type":"string", "analyzer":"<analyzer-name>"}
```

To see the mapping generated by Oracle NoSQL Database for a given index created in Elasticsearch, you can execute a command like the following from the command line of a host with network connectivity to one of the nodes in the Elasticsearch cluster (example: `esHost`):

```
curl –X GET 'http://esHost:9200/ondb.<store>.<table>.<index>/_mapping?
pretty'
```

For details on the sort of additional mapping parameters you can supply to Elasticsearch via the mapping specification, see Elasticsearch Mapping Parameters.

As a concrete example, suppose you have a table named `jokeTbl` in a store named `kvstore`, where the table consists of a field named `category` with values representing the categories under which jokes can fall, along with a field named `txt` that contains a `string` consisting of a joke that falls under the associated category. Suppose that when indexing the values stored under the `category` field, you want to index each word that makes up the category; but when indexing each joke, you want the word stems (or word roots) to be stored rather than the whole words. For example, if a joke contains the word "solipsistic", the stem of the word - "solipsist" – would actually be indexed (stored) rather than the whole word.

Since the Elasticsearch "standard" analyzer breaks up text into whole words, and the "english" analyzer stems words into their root form, you would use the "standard"

analyzer for the category field and the "english" analyzer for the `txt` field (assuming the jokes are written in English rather than some other language). Specifically, to create the Text Index, you would execute a command like the following from the Admin CLI:

```
kv-> execute 'CREATE FULLTEXT INDEX jokeIndx ON jokeTbl (
       category{"type":"string","analyzer":"standard"},
       txt{"type":"string","analyzer":"english"})';
```

Once the Text Index is created, you can then query the index by executing a `curl` command from the command line of a host with network connectivity to one of the nodes in the Elasticsearch cluster. For example,

```
curl –X GET 'http://<esHost>:9200/ondb.kvstore.jokeTbl.jokeIndx/_search?
pretty'
```

To see the mapping generated by Oracle NoSQL Database for the `jokeIndx` in the example above, you can execute a `curl` command like the following:

```
curl –X GET 'http://<esHost>:9200/ondb.kvstore.jokeTbl.jokeIndx/
_mapping?pretty'
```

> **Note:**
>
> Text indexed fields can include non-scalar types (such as map and array), which are specified in the same way, and with the same limitations, as those for Oracle NoSQL Secondary Indexes.

Handling `TIMESTAMP` Data Type

Topics

- Mapping Oracle NoSQL `TIMESTAMP` to Elasticsearch `date` Type
- Full Text Search of Indexed `TIMESTAMP` Scalar

Mapping Oracle NoSQL `TIMESTAMP` to Elasticsearch `date` Type

When a value representing a date and time is written to a field of an Oracle NoSQL table, the value is stored in the table as an instance of `java.sql.Timestamp`; which corresponds to the Oracle NoSQL `timestamp enum` type. See Atomic Data Types in the *SQL Reference Guide*.

When creating a table, the keyword `timestamp` is then used to specify such a field in the table. Along with the `timestamp` keyword, an `integer` parameter representing the `precision` to apply when storing the value must also be specified, employing a declaration with the following form:

```
TIMESTAMP(<precision>)
```

The value input for `precision` must be one of ten possible `integer` values, from 0 to 9. In general, the `timestamp` data type defined by Oracle NoSQL Database allows finer-grained time precisions to be stored in a table; up to nanosecond granularity. A value of 0 input for `precision` specifies the least precise representation of a `timestamp` value; which corresponds to a format of, `yyyy-MM-dd'T'HH:mm:ss`, with 0 decimal places in the value's seconds component. A value of 9 specifies the finest granularity - or most precise - representation, which includes an instant during the given day that is accurate to the nanosecond. `timestamp` values with nanosecond precision correspond to a format of `yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS`, with 9 decimal places in the seconds component. All other precisions (1–8) represent a day and time granularity falling somewhere between the least precise (0 decimal places) and the most precise (9 decimal places).

As another concrete example, suppose you wish to create a table named `tsTable` consisting of an `id` field containing the table's Primary Key, and a field named `ts` that will contain values representing a date and a time-of-day in which the seconds component is represented with 6 decimal point accuracy (example: `date = 1998-10-26, time-of-day = 08:33:59.735978`). To create such a table, one can execute the following command from the Admin CLI:

```
kv-> execute 'CREATE TABLE tsTable (id INTEGER, ts TIMESTAMP(6),
PRIMARY KEY (id))';
```

Suppose then that you wish to store the following values in the `ts` field:

```
tsVal[0] = 1996-12-31T23:01:43.987654321
tsVal[1] = 2005-03-20T14:10:25.258
tsVal[2] = 1998-10-26T08:33:59.735978
tsVal[3] = 2001-09-15T23:01:43.55566677
tsVal[4] = 2002-04-06T17:07:38.7653459
```

To store those values, you could execute code like the following:

```
final KVStore store = KVStoreFactory.getStore
    (new KVStoreConfig(<storeName>, <host> + ":" + <port>));
final TableAPI tableAPI = store.getTableAPI();
final Tabletable = tableAPI.getTable("tsTable");
for (int i = 0; i < 5; i++) {
    final Row row = table.createRow();
    row.put(id, i);
    row.put(ts, TimestampUtils.parseString(tsVal[i]));
    tableAPI.putIfAbsent(row, null, null);
}
```

Because the `ts` field of the table was created with precision 6, each value will be stored with 6 decimal places in the seconds component of the value. Specifically, if the value being stored contains more than 6 decimal places, then Oracle NoSQL will store the value with the decimal part of the seconds component rounded to 6 decimal places. For example, `tsVal[4]` from the list above will be stored as, `2002-04-06T17:07:38.765346`.

Similarly, if the value being stored contains fewer than 6 decimal places, then Oracle NoSQL will pad the decimal part of the seconds component with zeros. For example, `tsVal[1]` from the list above will be stored as, `2005-03-20T14:10:25.258000`.

When creating a Text Index on a table's field containing `timestamp` values, it is important to understand how the Oracle NoSQL Database Table API handles fields such as those described above. It is important because Elasticsearch stores values representing date and time using the Elasticsearch `date` type; which does not map directly to the `java.sql.Timestamp` type stored by Oracle NoSQL Database.

When indexing a `timestamp` field for Full Text Search, the Elasticsearch `date` type must be specified in the `CREATE FULLTEXT INDEX` command; otherwise Elasticsearch will handle the field's values as a `string` type. For example, the simplest way to index (for full text search) the `ts` field from the `tsTable` in the example above, would be to execute the following command:

```
kv-> execute 'CREATE FULLTEXT INDEX tsIndex ON tsTable
(ts{"type":"date"})';
```

In this case, a default mapping specification will be generated that will tell Elasticsearch to handle the broadest range of `date` type formats when handling the values being indexed.

When indexing values that represent date and time in Elasticsearch, whenever you specify the `date` type for those values, you can also specify a `format` to which each indexed value must adhere; where an error will occur if a given value does not satisfy the specified format. See Elasticsearch Date. In a fashion similar to how one specifies an "analyzer" for a "string" value, the Elasticsearch API defines a `format` parameter that can be used to specify – via the mapping specification – the format Elasticsearch should expect when indexing a given value of type `date`. Specifically,

```
<fieldname>{"type":"date","format":"<format>"}
```

where the value input for the `format` token can be an explicit format such as, `yyyy-MM-dd'T'HH:mm:ss`, or can be a combination one or more of the Elasticsearch pre-defined values (macros). See Elasticsearch Built In Formats.

Using the Elasticsearch API (not Oracle NoSQL), a typical Elasticsearch mapping specification for a `date` type might then specify an explicit format along with one or more values from the set of Elasticsearch built in formats; for example,

```
{"type":"date","format":"yyyy-MM-dd'T'HH:mm:ss.SSS||yyyy-MM-dd||
epoch_millis"}
```

A format like that shown tells Elasticsearch to expect values in a form such as, `1997-11-17T08:33:59.735`, or `1997-11-17`, or even as a number of milliseconds since the epoch. If a value has any other format, an error will occur and Elasticsearch will not index (store) the value.

Rather than employing an explicit format such as that shown in the example above, you can also specify formats using some combination of only the macros from the table; for example,

```
{"type":"date","format":"strict_date_optional_time||epoch_millis"}
```

This tells Elasticsearch that although acceptable date values must include the date (`strict_date=yyyy-MM-dd`), Elasticsearch should accept any values with or without a time component (`optional_time`). Additionally, if the value represents the number of milliseconds since the epoch, then such values should also be accepted by Elasticsearch.

With respect to using the `CREATE FULLTEXT INDEX` command to index a `timestamp` value for Full Text Search, although it is possible to specify the Elasticsearch `format` parameter for a `date` field in a way similar to the Elasticsearch API examples shown above, it may not be very practical. First, the number of valid combinations of macros from the set of Elasticsearch built in formats is very large, and may pose a significant burden for users.

Next, unlike other mapping parameters defined by Elasticsearch (for example the "analyzer" parameter for "string" types), if the user specifies a valid format for an Elasticsearch `date` field, but one or more of the values to be indexed do not satisfy that format, then an error will occur (in Elasticsearch) and those values will not be indexed. For example, if the user specifies a "french" analyzer for a `string` field but the value is actually in English, although unexpected search output may result, no error will occur. On the other hand, if the user specifies a format of `yyyy-MM-dd'T'HH:mm:ss.SSS` for a `date` field, but the value(s) being indexed contains more than 3 decimal places in the seconds component, although the index will be created, format errors will occur and the non-conformant values will not be indexed;.

To provide a more convenient mechanism for specifying the format for `date` values, as well as to minimize the opportunity for the sort of format errors just described, a special `"name":"value"` parameter is defined for the `CREATE FULLTEXT INDEX` command. When indexing Oracle NoSQL `timestamp` values as `date` values in Elasticsearch, rather than using the Elasticsearch `format` parameter (and its valid values), the specially defined `precision` parameter should be used instead. Although the `precision` parameter is optional, when it is included with a `"type":"date"` specification in the `CREATE FULLTEXT INDEX` command, the value of that parameter can be either `millis` or `nanos`. Specifically, when the `CREATE FULLTEXT INDEX` command is used to index NoSQL `timestamp` values as `date` values in Elasticsearch, one of the following parameter mappings must be specified in that command:

- `{"type":"date"}`
- `{"type":"date","precision":"millis"}`
- `{"type":"date","precision":"nanos"}`

Note that the default `precision` (that is, `no precision`), as well as the `nanos precision`, both map - and index - the broadest range of `timestamp` formats as valid `date` types in Elasticsearch without error; whereas the `millis precision` indexes only `timestamp` values defined with precision 3 or less. As a result, the precision you use should be based on the following criteria:

- If you know for sure that all values from the table field to be indexed have only precision 3 (milliseconds) or less, and you want to index the values using 3 decimal places in all cases, then specify `millis precision`.
- If the field you wish to index consists of `timestamp` values of varying precisions and you want to index only those values with precision 3 or less, then specify `millis precision`; so that values with greater than milliseconds precision will not be indexed.
- In all other cases, use either `nanos precision` or the default `precision`.

In summary, the special `precision` parameter not only minimizes the number of possible values the user can specify for the `date` type, it also reduces the occurrence of format errors by providing a way to map such values to the broadest range of possible formats; as well as allow the user to enforce milliseconds precision in the index.

> **Note:**
>
> As described above, a precision of `nanos` specified for a date type is currently identical to specifying no precision, which translates to the default `date` format. Although this may seem redundant, the `nanos` option is defined for two reasons. First, it is intended to be symmetric with the `millis` option; so that if a user knows the `timestamp` field being indexed consists of values with greater than millisecond precision, the user can simply specify `nanos` and the *right thing* will be done when constructing the mapping specification that will be registered with Elasticsearch.
>
> The second reason for defining the `nanos` option is related to the fact that Elasticsearch currently supports formats with precisions no greater than milliseconds. (Notice that the Elasticsearch built in formats include macros associated with nothing finer than `millis`). If a version of Elasticsearch is released in the future that supports formats including nanoseconds precision, then a fairly straightforward change can be made in Oracle NoSQL Database to map the current `nanos` option to the new format defined by Elasticsearch; requiring no change in the public api, and no change to user applications.

Full Text Search of Indexed `TIMESTAMP` Scalar

Suppose you start a store named `kvstore` and create the `tsTable` with the same `timestamp` values as those presented previously, where each such value was stored in the table with precision 6. After registering the store with your Elasticsearch cluster (running on a host named `eshost`), a Text Index named `tsIndex` on the table's `ts` field is created by executing the following command from the Admin CLI:

```
kv-> execute 'CREATE FULLTEXT INDEX tsIndex ON tsTable
(ts{"type":"date"})';
```

Executing queries such as the following can then be used to perform a Full Text Search on the data that was indexed:

List all values, sorted in ascending order

```
curl –X GET 'http://eshost:9200/ondb.kvstore.tstable.tsindex/_search?
pretty'
                  '-d {"sort":[{"ts":"asc"}]}'

{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  },
  "hits" : {
    "total" : 5,
    "max_score" : null,

    "hits" : [ {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0000",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"0"},
                         "ts":"1996-12-31T23:01:43.987654"},

      "sort" : [852073303123]

}, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0002",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"2"},
                         "ts":"1998-10-26T08:33:59.735978"},

      "sort" : [909435821111]

}, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0003",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"3"},
                         "ts":"2001-09-15T23:01:43.555667"},

      "sort" : [995911599555]

  }, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
```

```
      "_id" : "/w/0004",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"4"},
                         "ts":"2002-04-06T17:07:38.765346"},

      "sort" : [1024765658765]

}, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0001",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"1"},
                         "ts":"2005-03-20T14:10:25.258000"},

      "sort" : [1091264176173]

  } ]
  }
}
```

Perform an exact match to find a specific date and time

```
curl –X GET 'http://eshost:9200/ondb.kvstore.tstable.tsindex/_search?
pretty'
                  '-d {"query":{"term":
{"ts":"2005-03-20T14:10:25.258000"}}}'

{
  ....
  "hits" : {
    "total" : 1,
    "max_score" : null,

    "hits" : [ {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0001",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"1"},
                         "ts":" 2005-03-20T14:10:25.258000},

  } ]
  }
}
```

Find dates that fall within a specific range of dates and times

```
curl –X GET 'http://eshost:9200/ondb.kvstore.tstable.tsindex/_search?
pretty'
                  '-d {"query":{"range":{"ts":{"gte":"
```

```
1998-10-26T08:33:59.735978","lt":" 2002-04-06T17:07:38.9"}}}}'

{
  ....
  "hits" : {
    "total" : 3,
    "max_score" : null,

    "hits" : [ {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0004",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"4"},
                         "ts":"2002-04-06T17:07:38.765346},

}, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0002",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"2"},
                         "ts":"1998-10-26T08:33:59.735978"},

      "sort" : [909435821111]

}, {
      "_index" : "ondb.kvstore.tstable.ts",
      "_type" : "text_index_mapping",
      "_id" : "/w/0003",
      "_score" : null,

      "_source":{"_pkey":{"_table":"tstable","id":"3"},
                         "ts":"2001-09-15T23:01:43.555667"}

  } ]
  }
}
```

Handling JSON Data Type

Topics

- Review: Secondary Indexes on JSON Document Content
- Creating Text Indexes on JSON Document Content
- Full Text Search of Indexed JSON Documents

Review: Secondary Indexes on JSON Document Content

How to index, for Full Text Search, content from JSON documents stored in an Oracle NoSQL Database table is presented in the next section. But to help you better understand the material in that section, you should first review the material in Indexing JSON in the *SQL Reference Guide*. It describes how to store values in a field of a NoSQL table when those values consist of strings in valid JSON format; that is, when those values are JSON documents.

When reviewing those materials, it is important to not confuse creating a Secondary Index on JSON content with creating a Text Index. Creating a Text Index on a field containing JSON documents is presented in the next section of this document.

When JSON is stored in an Oracle NoSQL Database table, the *data can be any valid JSON, stored as a string*; referred to as a JSON document. Each such document stored in a field (or column) of a NoSQL table consists of elements that are referred to as either the fields or the attributes of the document. Thus, when discussing the elements of a given JSON document in the sections below, the term field and the term attribute can be used interchangeably; where the context should distinguish the field (or column) of an Oracle NoSQL table from the field (or attribute) of a JSON document stored in the table.

Although you can create a Secondary Index on the attributes of a JSON document stored in a given table, there are numerous restrictions on such indexes; restrictions which may make a Text Index more attractive. First, when creating a Secondary Index, you can only index the scalar attributes of the document. That is, the attributes cannot be nested JSON objects. Additionally, only `integer`, `long`, `double`, `number`, `string`, and `boolean` are supported Oracle NoSQL data types for JSON Secondary Indexes. Finally, you cannot perform Full Text Search on such an index.

For example, consider the following JSON document whose content specifies information related to a given member of the United States senate. For each senator (both current and former), a JSON document like that shown here is created and the Oracle NoSQL Table API can be used to store each such document in a column of a given table. Note that throughout this section and the following section, the example JSON document shown here will be referenced numerous times to demonstrate how such a JSON document can be indexed; in either a Secondary Index or a Text Index.

```
{
  "description": "Senior Senator for Ohio",
  "party": "Democrat",
  "congress_numbers": [223,224,225],
  "state": "OH",
  "startdate": "2010-01–03T05:04:09.456",
  "enddate": "2020-11-12T03:01:02.567812359",
  "seniority": 37,
  "current": true,
  "duties": {
    "committee": ["Ways and Means","Judiciary","Steering"],
    "caucus": ["Automotive","Human Rights","SteelIndustry"]
  },
  "personal": {
    "firstname":"Sherrod",
    "lastname":"Brown",
    "birthday":"1952-11-09",
```

```
    "social_media": {
      "website":"https://www.brown.senate.gov",
      "rss_url":"http://www.brown.senate.gov/rss/feeds",
      "twittered":"SenSherrodBrown"
    },
    "address": {
      "home": {
        "number":"9115-ext",
        "street":"Vaughan",
        "apt":null,
        "city":"Columbus",
        "state":"OH",
        "zipcode":43221,
        "phone": "614-742-8331"
      },
      "work": {
        "number":"Hart Senate Office Building",
        "street":"Second Street NE",
        "apt":713,
        "city":"Washington",
        "state":"DC",
        "zipcode":20001
        "phone": "202-553-5132"
      }
    },
    "cspanid": 57884
  },
  "contrib": 2571354.93
}
```

The example JSON document above consists of a variety of JSON attributes of different types. Some attributes are scalar fields in `"name":"value"` form, whereas others are either nested objects, or arrays of scalar values. An attribute that is a nested object is a structure, encapsulated by curly braces `{...}`, that contains a set of valid JSON field types; scalars, arrays of scalars, and/or JSON objects (named or unnamed). An array type is an ordered, comma-separated list of elements, encapsulated by square brackets `[...]`, where each element must be the same scalar type; `string`, `date`, or numerical type (`integer`, `double`, `number`, and so on).

The value of a scalar field nested within an object is dereferenced using JSON path notation. For example, the scalar field containing each senator's date of birth is nested in the object named `personal`. Each senator's birthday can then be specified in a search query using the JSON path, *jsonFieldName*.personal.birthday; where the value of the *jsonFieldName* component is the name specified for the column of the table in which each JSON document is written. Similarly, a search on each senator's home city can be expressed using the path, *jsonFieldName*.personal.address.home.city.

Note that in Elasticsearch, array fields are handled in a way that may be unexpected. When querying arrays in Elasticsearch, you cannot refer to the "first element", the "last element", the "element at index 3", etc. Arrays are handled as a "bag of values of the same type". For the example document above, if you wanted to search the committees on which each senator serves, you would not construct your query using a path like, *jsonFieldName*.duties.committee[0]. Such a path is not allowed. Instead, you would specify the path to the array itself, along with the

values you wish to search for that may be elements of the array; for example, "*jsonFieldName*.duties.committee":"Judiciary Steering".

As discussed previously, each attribute in a JSON document has a type; where the type is implied by the structure of the attribute, or the value associated with the attribute. An attribute in a JSON document whose content is encapsulated by curly braces implies that the attribute is a JSON object type. With respect to scalar fields, the implied data type of the value associated with such a field is dependent on the value of the field itself. This is true whether the index is a Secondary Index or a Text Index. For example, the scalar attributes named `description` and `seniority` from the JSON document shown above will be handled as `string` and `integer` types respectively.

Compare this with a value such as that specified for the JSON document's `contrib` attribute (`2571354.93`). Such a scalar value will be handled as a NoSQL `double` data type when creating a Secondary Index; and as either an Elasticsearch `float` or `double` type when creating a Text Index for Full Text Search in an Elasticsearch cluster. Similarly, for attributes that contain information representing date and time (example the `startdate`, `enddate`, and `birthday` attributes), the value of such fields can only be handled as an Oracle NoSQL `string` type when creating a Secondary Index, but may be handled as either an Elasticsearch `string` or `date` type when creating a Text Index.

Finally, although an attribute containing a comma-separated list of scalars encapsulated by square brackets implies a JSON array type, the data type of the array's elements (that is, the array's type) is implied by the values of the elements in the same way as was previously described for scalar attributes.

Suppose then that you wish to create a table named `jsonTable` consisting of an `id` field containing the table's Primary Key, and a field named `jsonField` that will contain values consisting of JSON documents like the example document presented previously. To create such a table, and examine its resulting structure, one would execute a command like the following from the Admin CLI:

```
kv-> execute 'CREATE TABLE jsonTable
         (id INTEGER, jsonField JSON, PRIMARY KEY (id))';

kv-> execute 'DESCRIBE AS JSON TABLE jsonTable';
{
  "json_version" " 1,
  "type" : "table",
  "name" : "jsonTable",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER",
    "nullable" : false,
    "default" : null
  }, {
    "name": "jsonField",
    "type" : "JSON",
    "nullable" : true,
    "default" : null
```

```
  } ]
}
```

To populate the table with JSON documents like the example document presented above, you could execute code like the following:

```
final KVStore store = KVStoreFactory.getStore
   (new KVStoreConfig(<storeName>, <host> + ":" + <port>));
final tableAPI = store.getTableAPI();
final table = tableAPI.getTable("tsTable");
final List<String> listOfJsonDocs = {...};
for (int i = 0; i < listOfJsonDocs.size(); i++) {
    final Row row = table.createRow();
    row.put(id, i);
    row.putJson("jsonField", listOfJsonDocs.get(i));
    tableAPI.putIfAbsent(row, null, null);
}
```

After populating the table with the necessary JSON documents (using the method `row.putJson` from the Table API), a Secondary Index on selected attributes of each document stored in the table's `jsonField` field can be created by executing a command like:

```
kv-> execute 'CREATE INDEX jsonSecIndex ON jsonTable
         (jsonField.party AS STRING,
          jsonField.current AS BOOLEAN,
          jsonField.contrib AS DOUBLE,
          jsonField.seniority AS INTEGER)';
```

In this case, queries can be performed based on various combinations of each senator's party affiliation, seniority, total amount of money contributed to the senator's campaign, and whether or not the senator is a currently sitting senator. For example, to find all current democratic senators with contributions totaling between 1 million and 20 million dollars, a command like the following could be executed from the Admin CLI:

```
kv-> GET TABLE –name jsonTable
  –index jsonSecIndex
  -field jsonField.party –value "Democrat"
  -field jsonField.current –value true
  -field jsonField.contrib –start 1000000.00 –end 20000000
```

Creating Text Indexes on JSON Document Content

Using the example presented previously, this section describes how to create a Text Index on the contents of a JSON document stored in a NoSQL table, and then perform various Full Text Search queries on the resulting index in Elasticsearch.

Unlike Oracle NoSQL Database Secondary Indexes, where the type of each value stored in a field of a given table is inferred from the table schema, for Text Indexes, the type of each attribute to be indexed cannot be inferred from the schema; and thus, must be specified in the `CREATE FULLTEXT INDEX` command. Although the table's

schema tells Oracle NoSQL that the values in a given field (column) of a table is a JSON document, it tells Oracle NoSQL nothing about the internal structure of the document itself, other than each element is JSON formatted content. Since Oracle NoSQL knows neither the attributes within the JSON document to be indexed, nor the data types that should be used when indexing those attributes, that information must be explicitly given to Oracle NoSQL via the `CREATE FULLTEXT INDEX` command.

Thus, to create a Text Index on a column containing JSON documents, in addition to specifying the attributes to index, in JSON path notation, you must also always provide a mapping specification. This tells Oracle NoSQL the attributes within the document to index, as well as the data type to tell Elasticsearch to use when indexing each such attribute.

For example, in the previous section a Secondary Index was created and queried to find all current democratic senators with contributions totaling between 1 million and 20 million dollars. But suppose you want to refine that search, to find all current democratic senators with contributions totaling between 1 million and 20 millions dollars, who also serve on either the Judiciary or Appropriations committee (or both). For such a search, a Text Index should be created instead of a Secondary Index; not only because the committee information is contained in a nested array of strings, but also so that a Full Text Search can be performed.

To do this, first create the desired Text Index by executing the following command from the Admin CLI:

```
kv-> execute 'CREATE FULLTEXT INDEX jsonTxtIndex ON
    jsonTable (
      jsonField.current{"type":"boolean"},
      jsonField.party{"type":"string","analyzer":"standard"},
      jsonField.duties.committe{"type":"string"},
      jsonField.contrib{"type":"double"})';
```

Rather than creating a Secondary Index on the ts column of the table named `jsonTable`, like you did in the previous section's example, the command above instead creates a Text Index consisting of specific attributes of the documents stored in that column. Although the previous example index allowed you to find all current democratic senators with contributions totaling between 1 million and 20 million dollars, the Text Index created above allows the search to be refined. With the Text Index, you can search for all current democratic senators with contributions totaling between 1 million and 20 millions dollars, who also serve on either the Judiciary or Appropriations committee, or both.

After creating the Text Index using the command above, you can then query Elasticsearch for the documents that satisfy the desired search criteria by executing a `curl` command from a node that has network access to the Elasticsearch cluster with which the Oracle NoSQL store is registered. For example, from the node named `esHost`,

```
curl –X GET
'http://esHost:9200/ondb.kvstore.jsontable.jsontxtindex/_search?pretty'
'-d {query":{"bool":{
"must":{"match":{"jsonField.party":"Democrat"}},
"must":{"match":"jsonField.current":"true"}},
"must":{"range":{"jsonField.contrib":{
        "gte":"1000000.00","lte":20000000.00"}}},
```

```
"must":"match":{"jsonField.duties.committe":
                 "Judiciary Apropriations"}}}}}'
```

As previously explained, `ondb.kvstore.jsontable.jsontxtindex` in the query above is the name of the index that Oracle NoSQL creates in Elasticsearch; where `kvstore` is the name of the Oracle NoSQL store, `jsontable` corresponds to the table (jsonTable) in that store that contains the JSON documents being indexed, and `jsontxtindex` corresponds to the Text Index metadata maintained by the store.

The output produced by the Elasticsearch query above (with some re-formatting for readability) should look something like:

```
{
  ....
  "hits" : {
    "total" : 31,
    "max_score" : 1.4695805,

    "hits" : [ {
      "_index" : "ondb.kvstore.jsontable.jsontindex",
      "_type" : "text_index_mapping",
      "_id" : "/w/0001",
      "_score" : 1.4695805,

      "_source":{"_pkey":{"_table":"jsontable","id":"1"},
        "jsonField":"{"description":
                        "Senior Senator for Ohio"},
        "jsonField"{"current":"true"},
        "jsonField":{"congress_numbers":[223,224,225]},
        "jsonField":{"party":"Democrat"},
        "jsonField":{"seniority":37},
        "jsonFeld":{"personal":{"birthday":1952-11-09"}},
        "jsonField":{"personal":{"lastname":"Brown"}},
        "jsonField":{"contrib":257134.93},
        "jsonField":{"duties":{"committee":["Ways and
            Means","Judiciary","Democratic Steering"]}},
        "jsonField":{"duties":{"caucus":["Congressional
            Automotive","Human Rights","Steel Industry"]}},
        "jsonField":{"personal":{"address":{"home":{
                                    "state":"OH"}}}},
        "jsonField":{":"personal":{"address":{"home":{
                                    "city":"Columbus"}}}}}
    } ],
    ....
  }
}
```

It is important to understand that unlike the query against the Secondary Index presented in the previous section, this query is executed against the Elasticsearch cluster rather than the Oracle NoSQL store. Additionally, the Text Index created here allows one to perform a Full Text Search on the values in the nested array `jsonField.duties.committee`; something that cannot be done with Secondary Indexes.

Full Text Search of Indexed JSON Documents

This section presents the steps to execute a simple but complete example, without security. Although in a production setting, both the Oracle NoSQL Database and the Elasticsearch cluster should generally be run on separate nodes, for simplicity, these steps are executed on a single node. Additionally, if you already have an Elasticsearch version 2 cluster running in your environment, then feel free to use that cluster in place of the Elasticsearch single-node cluster used below. Note finally, that you may have to change some of the `tokens` (directory locations, version numbers, etc.) to suit your particular environment.

1. Download, install, and run Elasticsearch, version 2.

   Download the tar file https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/tar/elasticsearch/2.4.6/elasticsearch-2.4.6.tar.gz and place it under the directory `/opt/es`.

   ```
   cd /opt/es
   tar xzvf elasticsearch-2.4.6.tar.gz
   ln –s elasticsearch-2.4.6 elasticsearch
   export JAVA_HOME=/usr/lib/jvm/java8
   /opt/es/elasticsearch/bin/elasticsearch
           -Dnetwork.host=localhost
           --cluster.name kv-es-cluster
           --node.name localhost
   ```

   > **Note:**
   >
   > Elasticsearch version 2 requires Java 8. Thus, you should install Java 8 and set the `JAVA_HOME` environment to point to the Java 8's home directory.

2. Use KVLite to deploy an Oracle NoSQL Database store named `kvstore`.

   Assuming that you have installed Oracle NoSQL Database under the directory `/opt/ondb`, and that you have write permission for your system's `/tmp` directory, execute the following command from a command line:

   ```
   java –jar /opt/ondb/kv/lib/kvstore.jar kvlite
     –root /tmp/kvroot
     –host localhost
     –port 5000
     –store kvstore
     –secure-config disable
   ```

3. Start the Oracle NoSQL Database Admin CLI.

   From a separate command window, execute the command:

   ```
   java –jar /opt/ondb/kv/lib/kvstore.jar runadmin
     –host localhost
   ```

```
  -port 5000
  -store kvstore
```

4. Install a file containing the JSON documents to load.

   Under a directory such as `~/examples/es/docs`, create a file named `senator-info.json` and populate it with one or more JSON documents like those shown in the example file presented in Sample: Array of JSON Documents. Be sure to format the file you create with the same format shown in Sample: Array of JSON Documents.

5. Compile and execute the `LoadJsonExample` program (or similar).

   Under a directory such as `~/examples/es/src`, create the sub-directory `es/table`, and then create a file named `LoadJsonExample.java` under the directory `~/examples/es/src/es/table`. After creating the file `~/examples/es/src/es/table/LoadJsonExample.java`, add the source code presented in The `LoadJsonExample` Program Source (or source with similar functionality).

   Once the `LoadJsonExample.java` program is created, execute the following from a separate command window:

```
cd ~/examples/es/src

javac -classpath /opt/ondb/kv/lib/kvstore.jar:src
  examples/es/table/LoadJsonExample.java

java -classpath /opt/ondb/kv/lib/kvstore.jar:src
  es.table.LoadJsonExample
  -store kvstore
  -host localhost
  -port 5000
  -file ~/examples/es/docs/senator-info.json
  -table exampleJsonTable
```

> **Note:**
>
> The source code for the `LoadJsonExample` program that is presented in The `LoadJsonExample` Program Source is only intended to provide a convenient mechanism for loading non-trivial JSON content into an Oracle NoSQL table. You should feel free to write your own program to provide similar functionality.

6. Create a Text Index on the JSON data loaded into the NoSQL table.

   After verifying that the table has been successfully created and populated with the desired table data, execute the following from the Admin CLI:

```
kv-> plan register-es
  -clustername kv-es-cluster
  -host localhost
  -port 9200
  -secure false
  -wait
```

```
kv-> execute 'CREATE FULLTEXT INDEX jsonTxtIndex ON
exampleJsonTable (
  jsonField.current{"type":"boolean"},
  jsonField.party{"type":"string","analyzer":"standard"},
  jsonField.duties.committe{"type":"string"},
  jsonField.contrib{"type":"double"})';
```

7. Execute Full Text Search queries against data indexed in Elasticsearch.

   To first verify that the desired index has been created in Elasticsearch as expected, execute the following from a command line:

   ```
   curl –X GET 'http://localhost:9200/_cat/indices'

   yellow open ondb.kvstore._checkpoint ...
   yellow open ondb.kvstore.examplejsontable.jsontxtindex ...
   ```

   Note that Elasticsearch reports the status of each index is yellow. This occurs here because the Elasticsearch cluster was deployed as a single-node cluster.

   To examine the mapping that Oracle NoSQL constructs for Elasticsearch, execute:

   ```
   curl –X GET 'http://localhost:9200/
   ondb.kvstore.examplejsontable.jsontxtindex/_mapping?pretty'
   ```

   To display all documents from the `exampleJsonTable` that were indexed in Elasticsearch, execute:

   ```
   curl –X GET 'http://localhost:9200/
   ondb.kvstore.examplejsontable.jsontxtindex/_search?pretty'
   ```

   Finally, to find all current democratic senators with contributions totaling between 5 million and 15 million dollars, who are members of either the "Progressive" caucus or the "Human Rights" caucus, execute the following command:

   ```
   curl –X GET
     'http://localhost:9200/ondb.kvstore.examplejsontable.jsontxtindex/
   _search?pretty'
     '-d {query":{"bool":{
       "must":{"match":{"jsonField.party":"Democrat"}},
       "must":{"match":"jsonField.current":"true"}},
       "must":{"range":{"jsonField.contrib":
   {"gte":"5000000.00","lte":15000000.00"}}},
       "must":"match":{"jsonField.duties.caucus":"Progressive Human
   Rights"}}}}}'
   ```

Deleting a Full Text Index

To delete a Full Text Index created on an Oracle NoSQL table, you can use the NoSQL store's Admin CLI to execute a command with the following format:

```
execute 'DROP INDEX [IF EXISTS] <index> ON <table>';
```

Each argument, flag, and directive is described as follows, where any item encapsulated by square brackets `[...]` is optional:

- `index` - The name of the Text Index to delete.
- `table` - The name of the table containing the indexed fields.

If the command above is executed and a Text Index with the specified name does not exist, then the command will fail, displaying an error message. Specifying `IF EXISTS` when the named index does not exist will result in a *no-op*.

> **Note:**
>
> The command above, when applied to a Full Text Index, will not only remove all metadata related to the index from the associated Oracle NoSQL store's state, but will also remove the corresponding data indexed in Elasticsearch.

18
Security in Full Text Search

Topics

Elasticsearch and Secure Oracle NoSQL Database

Up to this point, all information and examples presented in the previous sections discussed how data stored in an Oracle NoSQL Database table is indexed in Elasticsearch when the communication between Oracle NoSQL Database and Elasticsearch is not secure. This section discusses how that data can be sent to the Elasticsearch cluster over a *secure* communication channel.

As described previously, data sent to Elasticsearch for indexing is sent by a process running on the master replication node of the Oracle NoSQL store's replication group (or shard). When the system is not configured for security, the replication node communicates with Elasticsearch over HTTP. For the replication node to send the data to Elasticsearch over a secure communication channel, the NoSQL store must be configured to run securely. See Introducing Oracle NoSQL Database Security in the *Security Guide*. When configured for secure communication, the replication node will send the data to Elasticsearch, in encrypted form, over HTTPS. This means that Elastisticsearch must be configured to perform the necessary authentication and decryption before indexing the data received from a secure Oracle NoSQL store.

Elasticsearch version 2 does not provide a fully integrated, out-of-the-box option for communicating with clients over a secure channel in the manner just described. For secure communication with Elasticsearch, some users choose to run their Elasticsearch deployment "behind" (or "wrapped" within) a secure web server. Others choose to employ one of the commercially available plugins that support TLS (SSL) for this purpose. Oracle NoSQL Database has chosen to support the latter model.

In order to communicate securely with the Elasticsearch cluster, Oracle NoSQL Database recommends that the Shield proprietary plugin be used to provide a port to which clients of the Elasticsearch cluster can connect and communicate securely over HTTPS.

> **Note:**
>
> Although the Shield plugin has been used when testing secure communication between the current Oracle NoSQL Database implementation and Elasticsearch version 2, there is nothing in the NoSQL implementation that should prevent the use of other such Elasticsearch security plugins; as long as the plugin supports HTTPS, and can be configured to support the Oracle NoSQL Database authentication scheme.

Compared to the non-secure case presented previously, there are additional steps you must take when working with the secure case. For the secure case, the Oracle NoSQL store will be populated using the secure mode of the same example program, and the indexed data will be queried using similar queries, as that presented for the non-secure case. The only difference is that the Oracle NoSQL store and the Elasticsearch cluster will each be deployed to communicate securely, and the queries will specify the necessary keys and certificates required by the Elasticsearch cluster.

Deploying a secure Oracle NoSQL store and Elasticsearch cluster and configuring them to communicate securely with each other requires many more steps than the non-secure case. Appendices Secure Elasticsearch using Sheild, Deploying and Configuring a Secure Oracle NoSQL Store, and Install the Full Text Search Public Certificate in Elasticsearch provide detailed descriptions of all the steps necessary to deploy such a system. And once you have successfully deployed a secure Oracle NoSQL store and a secure Elasticsearch cluster, and you have installed the necessary artifacts (certificates) for the store and cluster to communicate, there are only minor differences between the commands and queries presented previously for the non-secure case and their counterparts in the secure case.

One of the first differences to note is that when executing the `LoadJsonExample` program to populate the NoSQL store with data to index in Elasticsearch, you must specify the `security` parameter with the absolute path to the file containing the login properties required by Oracle NoSQL Database Security (see Deploying and Configuring a Secure Oracle NoSQL Store for details). For example,

```
java –classpath /opt/ondb/kv/lib/kvstore.jar:src
es.table.LoadJsonExample
        –store kvstore
        –host localhost
        –port 5000
        –file ~/examples/es/docs/senator-info.json
        -table exampleJsonTable
        -security /tmp/FTS-client.login
```

Next, when executing the `register-es` command to register the NoSQL store with the secure Elasticsearch cluster, you must specify true for that command's `secure` parameter. For example,

```
kv-> plan register-es
        –clustername escluster
        –host eshost1
        –port 29100
        –secure true
        -wait
```

Finally, when querying the data indexed by the secure Elasticsearch cluster, the `curl` command must include the OpenSSL public certificate and private key required by the cluster for authentication of the request. See Secure Elasticsearch using Sheild. For example,

```
curl –k -E /tmp/elasticsearch-eshost1.pem
        --key /tmp/elasticsearch-eshost1.pkey
        –X GET 'http://eshost1:29100/
ondb.kvstore.jsontable.jsontxtindex/_search?pretty'
```

```
        '-d {query":{"bool":{
                "must":{"match":{"jsonField.party":"Democrat"}},
                "must":{"match":"jsonField.current":"true"}},
                "must":{"range":{"jsonField.contrib":
{"gte":"1000000.00","lte":20000000.00"}}},
                "must":"match":{"jsonField.duties.committe":"Judiciary
Apropriations"}}}}}'
```

With respect to secure Full Text Search and the example commands presented in this document, it is assumed you have followed the directions presented in Secure Elasticsearch using Sheild, Deploying and Configuring a Secure Oracle NoSQL Store, and Install the Full Text Search Public Certificate in Elasticsearch appendices; which, for clarity and convenience, organize the steps to configure and deploy a secure Elasticsearch and Oracle NoSQL system into separate, self-contained sections.

Secure Elasticsearch using Sheild presents the steps required to configure Elasticsearch for security. These steps must be taken whether the Elasticsearch cluster will be communicating with a secure Oracle NoSQL store or some other service or client unrelated to Oracle NoSQL.

Deploying and Configuring a Secure Oracle NoSQL Store describes how to deploy a secure Oracle NoSQL store and then configure it to communicate securely with the Elasticsearch cluster described in Secure Elasticsearch using Sheild.

The final steps required to complete the deployment of the secure Oracle NoSQL and Elasticsearch system are presented in Install the Full Text Search Public Certificate in Elasticsearch. Those steps will complete the security configuration of the Elasticsearch cluster from Secure Elasticsearch using Sheild, and are required for the nodes of the cluster to communicate with the secure Oracle NoSQL store from Deploying and Configuring a Secure Oracle NoSQL Store. The steps presented in Install the Full Text Search Public Certificate in Elasticsearch should be executed only after executing the steps in Secure Elasticsearch using Sheild and Deploying and Configuring a Secure Oracle NoSQL Store.

After completing the steps presented in Secure Elasticsearch using Sheild, Deploying and Configuring a Secure Oracle NoSQL Store and Install the Full Text Search Public Certificate in Elasticsearch appendices, you should then be able to run the example program `LoadJsonExample` to populate a table in the secure Oracle NoSQL store deployed in Deploying and Configuring a Secure Oracle NoSQL Store, index data from that table in the secure Elasticsearch cluster from Secure Elasticsearch using Sheild and Install the Full Text Search Public Certificate in Elasticsearch, and finally run secure queries against the indexed data. For convenience, the secure versions of example commands you can execute are presented in Running the Examples in Secure Mode.

> **Note:**
>
> Unlike the non-secure example presented previously, instead of using KVLite to deploy an Oracle NoSQL store on a single node, Secure Elasticsearch using Sheild, Deploying and Configuring a Secure Oracle NoSQL Store, Install the Full Text Search Public Certificate in Elasticsearch, and Running the Examples in Secure Mode appendices show how to work with a secure Oracle NoSQL store and Elasticsearch cluster where both consist of three nodes rather than a single node. This is done to present a more realistic example, to demonstrate what one might typically encounter in production.

19
Appendix

Topics

Sample: Array of JSON Documents

The following sample file is in the format and content that is required by the `LoadJsonExample` program.

```
{
  "meta": {
    "limit": 2,
    "total_count": 2
  },
  "objects": [
  {
    "description": "Senior Senator for Ohio",
    "party": "Democrat",
    "congress_numbers": [223,224,225],
    "state": "OH",
    "startdate": "2010-01—03T05:04:09.456",
    "enddate": "2020-11-12T03:01:02.567812359",
    "seniority": 37,
    "current": true,
    "duties": {
      "committee": ["Ways and
                    Means","Judiciary","Steering"],
      "caucus": ["Automotive",
                 "Human Rights","SteelIndustry"]
    },
    "personal": {
      "firstname":"Sherrod",
      "lastname":"Brown",
      "birthday":"1952-11-09",
      "social_media": {
        "website":"https://www.brown.senate.gov",
        "rss_url":"http://www.brown.senate.gov/rss/feeds",
        "twittered":"SenSherrodBrown"
      },
```

```
      "address": {
        "home": {
          "number":"9115-ext",
          "street":"Vaughan",
          "apt":null,
          "city":"Columbus",
          "state":"OH",
          "zipcode":43221,
          "phone": "614-742-8331"
        },
        "work": {
          "number":"Hart Senate Office Building",
          "street":"Second Street NE",
          "apt":713,
          "city":"Washington",
          "state":"DC",
          "zipcode":20001
          "phone": "202-553-5132"
        }
      },
      "cspanid": 57884
    },
    "contrib": 2571354.93
  },
  {
    "description": "Junior Senator for Wisconsin",
    "party": "Indpendent",
    "congress_numbers": [113,114,115],
    "state": "WI",
    "startdate": "2013-01–03T03:02:01.123",
    "enddate": "2017-01-03T01:02:03.123456789",
    "seniority": 29,
    "current": true,
    "duties": {
      "committee": ["Intelligence","Judiciary",
                    "Appropriations"],
      "caucus": ["Congressional Progressive","Afterschool"]
    },
    "personal": {
      "firstname":"Tammy",
      "lastname":"Baldwin",
      "birthday":"1962-02-11",
      "social_media": {
       "website":"https://www.baldwin.senate.gov",
       "rss_url":"http://www.baldwin.senate.gov/rss/feeds",
       "twittered":"SenBaldwin"
      },
      "address": {
        "home": {
          "number":"23315",
          "street":"Wallbury Court",
          "apt":"17",
          "city":"Madison",
          "state":"WI",
          "zipcode":53779,
```

```
          "phone": "608-742-8331"
        },
        "work": {
          "number":"Hart Senate Office Building",
          "street":"Second Street NE",
          "apt":355,
          "city":"Washington",
          "state":"DC",
          "zipcode":20001
          "phone": "202-224-2315"
        }
      },
      "cspanid": 57884
    },
    "contrib": 2571354.93
  } ]
}
```

> **Note:**
>
> The `meta` object at the beginning of the file is required. The `meta` object has the `limit` and `total_count` equal to the number of JSON object elements in the objects array. Programs that read and load each JSON document will use the contents of that object to determine the total number of JSON documents contained in the file; specifically, the `limit` and the `total_count` attributes of the `meta` object. If you add additional documents to this example file, then update the values of the `meta` object accordingly.

The `LoadJsonExample` Program Source

The following `LoadJsonExample` java program creates and populates an Oracle NoSQL Database table with rows whose elements are JSON documents read from a text file.

```
package es.table;

import java.nio.file.Path;
import java.nio.file.Files;
import java.nio.file.FileSystems;

import java.io.FileNotFoundException;
import java.io.IOException;

import java.util.List;
import java.util.ArrayList;

import oracle.kv.FaultException;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.StatementResult;
```

```
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 com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.JsonToken;

import oracle.kv.impl.tif.esclient.jsonContent.ESJsonUtil;

/**
 * Class that creates an example table in a given Oracle NoSQL Database
store and
 * then uses the Oracle NoSQL Database Table API to populate the table
with
 * sample records consisting of JSON documents retrieved from a file.
The
 * table that is created consists of only two Oracle NoSQL Database
data types: an
 * INTEGER type and a JSON type.
 *
 * The file from which the desired JSON documents are retrieved must be
of
 * form:
 * <pre>
 {
    "meta": {
        "limit": n,
        "offset": 0,
        "total_count": n
    },
    "objects": [
        {
            JSON DOCUMENT 1
        },
        {
            JSON DOCUMENT 2
        },
        ....
        {
            JSON DOCUMENT n
        },
    ]
 }
 * </pre>
 */
public final class LoadJsonExample {

    final boolean debugWithNoStore = false;
    final boolean debugAll = false;
    final boolean debugTopLevelJsonArrayObject = false;
    final boolean debugAddDoc = false;
    final boolean debugJsonToStringTop = false;
```

```
final boolean debugJsonToStringArray = false;
final boolean debugDocByDoc = false;

private final KVStore store;
private final TableAPI tableAPI;
private final Table table;

private Path jsonPath;

private boolean deleteExisting = false;

private static final String TABLE_NAME_DEFAULT = "jsonTable";
private static final String ID_FIELD_NAME = "id";
private static final String JSON_FIELD_NAME = "jsonField";

public static void main(final String[] args) {
    try {
        final LoadJsonExample loadData = new LoadJsonExample(args);
        loadData.run();
    } catch (FaultException e) {
        e.printStackTrace();
        System.out.println("Please make sure a store is running.");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * Parses command line args and opens the KVStore.
 */
private LoadJsonExample(final String[] argv) {

    String storeName = "";
    String hostName = "";
    String hostPort = "";

    final int nArgs = argv.length;
    int argc = 0;
    String tableName = null;

    if (nArgs == 0) {
        usage(null);
    }

    while (argc < nArgs) {
        final String thisArg = argv[argc++];

        if ("-store".equals(thisArg)) {
            if (argc < nArgs) {
                storeName = argv[argc++];
            } else {
                usage("-store requires an argument");
            }
        } else if ("-host".equals(thisArg)) {
            if (argc < nArgs) {
```

```
                    hostName = argv[argc++];
                } else {
                    usage("-host requires an argument");
                }
            } else if ("-port".equals(thisArg)) {
                if (argc < nArgs) {
                    hostPort = argv[argc++];
                } else {
                    usage("-port requires an argument");
                }
            } else if ("-file".equals(thisArg)) {
                if (argc < nArgs) {
                    jsonPath =
FileSystems.getDefault().getPath(argv[argc++]);
                } else {
                    usage("-file requires an argument");
                }
            } else if ("-table".equals(thisArg)) {
                if (argc < nArgs) {
                    tableName = argv[argc++];
                }
            } else if ("-security".equals(thisArg)) {
                if (argc < nArgs) {
                    System.setProperty(
                        KVSecurityConstants.SECURITY_FILE_PROPERTY,
                        argv[argc++]);
                } else {
                    usage(“-security requires an argument”);
                }

            } else if ("-delete".equals(thisArg)) {
                deleteExisting = true;
            } else {
                usage("Unknown argument: " + thisArg);
            }
        }

        if (storeName == null) {
            usage("Missing argument: -store <store-name>");
        }

        if (hostName == null) {
            usage("Missing argument: -host <host>");
        }

        if (hostPort == null) {
            usage("Missing argument: -port <port>");
        }

        if (jsonPath == null) {
            usage("Missing argument: -file <path-to-file>");
        }

        /* When the table name is not specified, construct the name of
         * the table from the file name, minus the suffix. That is strip
```

```
             * off the path and the suffix and use file name's 'base' as
             * the name of the table.
             */
            if (tableName == null) {
                final Path flnmElement = jsonPath.getFileName();
                if (flnmElement == null) {
                    tableName = TABLE_NAME_DEFAULT;
                } else {
                    final String tmpTblName = flnmElement.toString();
                    final String suffixDelim = ".";
                    if (tmpTblName.contains(suffixDelim)) {
                        final int suffixIndx =
tmpTblName.indexOf(suffixDelim);
                        if (suffixIndx > 0) {
                            tableName = tmpTblName.substring(0, suffixIndx);
                        } else {
                            tableName = tmpTblName;
                        }
                    } else {
                        tableName = tmpTblName;
                    }
                }
            }


System.out.println("\n-----------------------------------------");
            System.out.println("Table to create and load = " + tableName);
            System.out.println("-----------------------------------------
\n");

            if (debugWithNoStore) {
                store = null;
                tableAPI = null;
                table = null;
            } else {
                store = KVStoreFactory.getStore
                    (new KVStoreConfig(storeName, hostName + ":" +
hostPort));

                tableAPI = store.getTableAPI();
                createTable(tableName);
                table = tableAPI.getTable(tableName);
                if (table == null) {
                    final String msg =
                        "Store does not contain table [name=" + tableName +
"]";
                    throw new RuntimeException(msg);
                }
            }
        }

        private void usage(final String message) {
            if (message != null) {
                System.out.println("\n" + message + "\n");
            }
```

```
        System.out.println("usage: " + getClass().getName());
        System.out.println
            ("\t-store <instance name>\n" +
             "\t-host <host name>\n" +
             "\t-port <port number>\n" +
             "\t[-file <path to file with json objects to add to table>
|\n" +
             "\t[-table <name of table to create and load>]\n" +
             "\t-delete (default: false) [delete all existing data
first]\n");
        System.exit(1);
    }

    private void run() throws FileNotFoundException, IOException {
        if (deleteExisting) {
            deleteExistingData();
        }
        doLoad(jsonPath);
    }

    private void createTable(final String tableName) {

        final String statement =
            "CREATE TABLE IF NOT EXISTS " + tableName +
            " (" +
               ID_FIELD_NAME + " INTEGER," +
               JSON_FIELD_NAME + " JSON," +
            "PRIMARY KEY (" + ID_FIELD_NAME + "))";

        try {
            final StatementResult result = store.executeSync(statement);
            if (result.isSuccessful()) {
                System.out.println("table created [" + tableName + "]");
            } else if (result.isCancelled()) {
                System.out.println("table creation CANCELLED [" +
                                   tableName + "]");
            } else {
                if (result.isDone()) {
                    System.out.println("table creation FAILED:\n\t" +
                                       statement);
                    System.out.println("ERROR:\n\t" +
                                       result.getErrorMessage());
                } else {
                    System.out.println("table creation IN
PROGRESS:\n\t" +
                                       statement);
                    System.out.println("STATUS:\n\t" +
result.getInfo());
                }
            }
        } catch (IllegalArgumentException e) {
            System.out.println("Invalid statement:");
            e.printStackTrace();
        } catch (FaultException e) {
```

```
            System.out.println("Failure on statement execution:");
            e.printStackTrace();
        }
    }

    private void doLoad(final Path filePath)
                    throws FileNotFoundException, IOException {

        try {
            loadJsonDocs(filePath);
        } finally {
            if (store != null) {
                store.close();
            }
        }
    }

    private void loadJsonDocs(final Path filePath)
                    throws FileNotFoundException, IOException {

        try {
        final byte[] jsonBytes = Files.readAllBytes(filePath);
            final int nDocsToAdd = getIntFieldValue(
                "meta", "total_count",
ESJsonUtil.createParser(jsonBytes));

            if (debugAll || debugTopLevelJsonArrayObject) {

                System.out.println("BEGIN Top Level Array Object:
'objects'");

                final String objectsJsonStr =
                    getTopLevelJsonArrayObject(
                        "objects", ESJsonUtil.createParser(jsonBytes));

                System.out.println(objectsJsonStr);
                System.out.println("END Top Level Array Object:
'objects'");
                System.out.println("\nBEGIN Top Level Array Object:
'meta'");

                final String metaJsonStr =
                    getTopLevelJsonArrayObject(
                        "meta", ESJsonUtil.createParser(jsonBytes));

                System.out.println(metaJsonStr);
                System.out.println("END Top Level Array Object:
'meta'");

            } /* endif (debugAll || debugTopLevelJsonArrayObject) */

            final String[] jsonArray =
                getJsonArrayElements(
                    "objects", ESJsonUtil.createParser(jsonBytes),
nDocsToAdd);
```

```
            for (int i = 0; i < nDocsToAdd; i++) {

                if (debugAll || debugAddDoc) {
                    System.out.println("Adding JSON Row[" + i +
                                       "] to table:\n" + jsonArray[i]);
                }
                addDoc(i, jsonArray[i]);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found [" +
                               filePath.getFileName() + "]: " + e);
            throw e;
        } catch (IOException e) {
            System.out.println("IOException [file=" +
                               filePath.getFileName() + "]: " + e);
            throw e;
        }
    }

    private void addDoc(final Integer id, final String jsonDoc) {

        final Row row = table.createRow();

        row.put(ID_FIELD_NAME, id);
        row.putJson(JSON_FIELD_NAME, jsonDoc);

        tableAPI.putIfAbsent(row, null, null);
    }

    private void deleteExistingData() {

        /* Get an iterator over all the primary keys in the table. */
        final TableIterator<PrimaryKey> itr =
            tableAPI.tableKeysIterator(table.createPrimaryKey(), null,
null);

        /* Delete each row from the table. */
        long cnt = 0;
        while (itr.hasNext()) {
            tableAPI.delete(itr.next(), null, null);
            cnt++;
        }
        itr.close();
        System.out.println(cnt + " records deleted");
    }

    /*
     * Convenience method for displaying output when debugging.
     */
    private void displayRow(Table tbl) {
        final TableIterator<Row> itr =
            tableAPI.tableIterator(tbl.createPrimaryKey(), null, null);
        while (itr.hasNext()) {
            System.out.println(itr.next());
```

```
        }
        itr.close();
    }


    /*
     * Supporting methods for parsing the various attributes in the
given file.
     */

    private String[] getJsonArrayElements(final String arrayName,
                                          final JsonParser parser,
                                          final int nElements)
                                              throws IOException {

        final List<String> arrayList = new ArrayList<String>();

        JsonToken token = parser.nextToken();
        while (token != null) {
            final String curFieldName = parser.getCurrentName();

            if (!arrayName.equals(curFieldName)) {
                token = parser.nextToken();
                continue;
            }
            break;
        }

        if (debugAll || debugDocByDoc) {
            System.out.println(
                "getJsonArrayElements loop: curToken = " + token +
                ", curFieldName = " + parser.getCurrentName());
        }

        token = parser.nextToken();

        if (debugAll || debugDocByDoc) {
            System.out.println(
                "getJsonArrayElements loop: nextToken = " + token +
                ", nextFieldName = " + parser.getCurrentName());
        }

        if (token == null) {
            System.out.println("getJsonArrayElements loop: " +
                               "*** WARNING - null first token from
parser");

            return arrayList.toArray(new String[nElements]);
        }

        if (token != JsonToken.START_ARRAY) {
            System.out.println("getJsonArrayElements loop: *** WARNING
- " +
                "first token from parser != " + "START_ARRAY [" + token
+ "]");
```

```
            return arrayList.toArray(new String[nElements]);
        }

        for (int i = 0; i < nElements; i++) {
            final StringBuilder strBldr = new StringBuilder();
            if (i > 0) {
                strBldr.append("{\n");
            }
            final String arrayElement =
                jsonToString(arrayName, null, parser, strBldr, false);
            arrayList.add(arrayElement);

            if (debugAll || debugDocByDoc) {
                System.out.println(
                    "getJsonArrayElements loop: arrayElement[" + i + "]
=\n" +
                    arrayElement);
            }

        }/* end loop */

        return arrayList.toArray(new String[nElements]);
    }

    private String getTopLevelJsonArrayObject(
                      final String fieldName,
                      final JsonParser parser) throws IOException {

        final StringBuilder strBldr = new StringBuilder();
        JsonToken token = parser.nextToken();
        while (token != null) {
            final String curFieldName = parser.getCurrentName();

            if (!fieldName.equals(curFieldName)) {
                token = parser.nextToken();
                continue;
            }
            break;
        }
        final String jsonStr =
            jsonToString(fieldName, null, parser, strBldr, false);
        return strBldr.toString();
    }

    private String jsonToString(final String stopField,
                                final JsonToken prevToken,
                                final JsonParser parser,
                                final StringBuilder strBldr,
                                final boolean fromObjectArray)
                                                    throws IOException {

        JsonToken token = parser.nextToken();

        if (token == null) {
```

```
            if (debugAll || debugJsonToStringTop) {
                System.out.println("TOP of jsonToString: prevToken = " +
                                   prevToken + ", curToken = " + token +
                                   ", RETURNING because input token ==
null");
            }
            return strBldr.toString();
        }

        final String curFieldName = parser.getCurrentName();

        if (debugAll || debugJsonToStringTop) {

            System.out.println("\nTOP of jsonToString");
            System.out.println("prevToken = " + prevToken +
                               ", curToken = " + token +
                               ", curFieldName = " + curFieldName +
                               ", stopField = " + stopField + ",
isScalar = " +
                               token.isScalarValue() + ",
fromObjectArray = " +
                               fromObjectArray);
        }

        if (prevToken != null) {

            if (prevToken == JsonToken.END_ARRAY ||
                prevToken == JsonToken.END_OBJECT) {

                if (debugAll || debugJsonToStringTop) {
                    System.out.println(
                        "*** TOP of jsonToString: prevToken != null [" +
                        prevToken + "] && " + "[END_ARRAY ||
END_OBJECT]");
                }

                if (stopField != null &&
stopField.equals(curFieldName)) {

                    if (token == JsonToken.END_ARRAY) {

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** STOP-BY-FIELD_NAME [END_ARRAY]: " +
                                "prevToken = " + prevToken + ",
curToken = " +
                                token + ", curFieldName = " +
curFieldName +
                                ", stopField = " + stopField +
                                " ... RETURN string, do not recurse, " +
                                "do not terminate outer array with
']' ..." +
                                " RETURN STR =\n" + strBldr.toString());
                        }
```

```
                        return strBldr.toString();
                    }

                    if (token == JsonToken.END_OBJECT) {

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** STOP-BY-FIELD_NAME [END_OBJECT]: "
+
                                "prevToken = " + prevToken + ",
curToken = " +
                                token + ", curFieldName = " +
curFieldName +
                                ", stopField = " + stopField +
                                " ... RETURN string, do not recurse, " +
                                "but DO terminate outer array with
'}' ..." +
                                " RETURN STR =\n" + strBldr.toString());
                        }
                        strBldr.append("\n}");
                        return strBldr.toString();
                    }
                } /* endif (STOP-BY-FIELD) */

                if (prevToken == JsonToken.END_OBJECT &&
                    token == JsonToken.START_OBJECT &&
                    curFieldName == null) {

                    if (fromObjectArray) {
                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** END_OBJECT && START_OBJECT && " +
                                "curFieldName=null && is OBJECT_ARRAY:
" +
                                "prevToken = " + prevToken + ",
curToken = " +
                                token + ", curFieldName = " +
curFieldName +
                                ", stopField = " + stopField +
                                ", fromObjectArray = " +
fromObjectArray +
                                " ... add ',{' and RECURSE");
                        }
                        strBldr.append(",\n{\n");
                        return jsonToString(
                          stopField, token, parser, strBldr,
fromObjectArray);
                    }

                    if (debugAll || debugJsonToStringTop) {
                        System.out.println(
                            "*** END_OBJECT && START_OBJECT && " +
                            "curFieldName=null && not OBJECT_ARRAY: " +
                            "prevToken = " + prevToken + ", curToken =
" +
```

```
                                token + ", curFieldName = " + curFieldName +
                                ", stopField = " + stopField +
                                ", fromObjectArray = " + fromObjectArray +
                                " ... simply return string, do not
recurse");
                    }
                    return strBldr.toString();

                } else if (prevToken == JsonToken.END_OBJECT &&
                           token == JsonToken.END_ARRAY &&
                           curFieldName != null) {

                    if (fromObjectArray) {

                        strBldr.append("\n]\n");

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** END_OBJECT && END_ARRAY && " +
                                "curFieldName != null && is
OBJECT_ARRAY: " +
                                "prevToken = " + prevToken + ",
curToken = " +
                                token + ", curFieldName = " +
curFieldName +
                                ", stopField = " + stopField +
                                " terminate OBJECT_ARRAY with ']' ... "
+
                                "do NOT RECURSE ... RETURN subString
=\n" +
                                strBldr.toString());
                        }

                    } else { /* NOT fromObjectArray */

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** END_OBJECT && END_ARRAY && " +
                                "curFieldName != null && NOT
OBJECT_ARRAY: " +
                                "prevToken = " + prevToken + ",
curToken = " +
                                token + ", curFieldName = " +
curFieldName +
                                ", stopField = " + stopField +
                                " do NOT terminate OBJECT_ARRAY ... " +
                                "do NOT RECURSE ... RETURN subString
=\n" +
                                strBldr.toString());
                        }

                    } /* endif (fromObjectArray) */

                    return strBldr.toString();
```

```
                } else if (prevToken == JsonToken.END_OBJECT &&
                           token == JsonToken.END_OBJECT &&
                           (curFieldName != null || fromObjectArray)) {

                    strBldr.append("}\n");

                    if (debugAll || debugJsonToStringTop) {
                        System.out.println(
                            "*** END_OBJECT && END_ARRAY && " +
                            "curFieldName != null OR is OBJECT_ARRAY: "
+
                            "prevToken = " + prevToken + ", curToken =
" +
                            token + ", curFieldName = " + curFieldName +
                            ", stopField = " + stopField +
                            " , fromObjectArray = " + fromObjectArray +
                            " terminate object in OBJECT_ARRAY with '}'
and " +
                            "RECURSE ...");
                    }
                    return jsonToString(
                          stopField, token, parser, strBldr,
fromObjectArray);

                } else if (prevToken == JsonToken.END_ARRAY &&
                           token == JsonToken.END_OBJECT &&
                           curFieldName != null) {

                    strBldr.append("}\n");

                    if (debugAll || debugJsonToStringTop) {
                        System.out.println(
                            "*** END_ARRAY then END_OBJECT && " +
                            "curFieldName != null: " +
                            "prevToken = " + prevToken + ", curToken =
" +
                            token + ", curFieldName = " + curFieldName +
                            ", stopField = " + stopField +
                            " , fromObjectArray = " + fromObjectArray +
                            " terminate object with '}' and
RECURSE ...");
                    }
                    return jsonToString(
                          stopField, token, parser, strBldr,
fromObjectArray);

                } else { /* DEFAULT: all other cases */

                    strBldr.append(",\n");

                    if (debugAll || debugJsonToStringTop) {
                        System.out.println(
                            "*** ELSE BLOCK *** prevToken = " +
prevToken +
                            ", curToken = " + token + ", curFieldName =
```

```
" +
                            curFieldName + ", stopField = " + stopField
+
                            " , fromObjectArray = " + fromObjectArray +
                            " add COMMA and RECURSE ...\n" +
                            strBldr.toString());
                    }
                    return jsonToString(
                          stopField, token, parser, strBldr,
fromObjectArray);

                } /* endif (prevToken, inputToken, curFieldName */

            } /* endif (prevToken == END_ARRAY || END_OBJECT) */

            if (prevToken.isScalarValue()) {

                if (token != JsonToken.END_ARRAY &&
                    token != JsonToken.END_OBJECT) {

                    strBldr.append(",\n");
                    return jsonToString(
                          stopField, token, parser, strBldr,
fromObjectArray);
                }

                if (stopField != null &&
stopField.equals(curFieldName)) {

                    if (token == JsonToken.END_ARRAY) {

                        strBldr.append("\n]");

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
                                "*** prev SCALAR && STOP-BY-FIELD_NAME
" +
                                "[END_ARRAY]: prevToken = " + prevToken
+
                                ", curToken = " + token + ",
curFieldName = " +
                                curFieldName + ", stopField = " +
stopField +
                                " ... terminate with ']' and RETURN
string");
                        }
                        return strBldr.toString();
                    }

                    if (token == JsonToken.END_OBJECT) {

                        strBldr.append("\n}");

                        if (debugAll || debugJsonToStringTop) {
                            System.out.println(
```

```
                                "*** prev SCALAR && STOP-BY-FIELD_NAME
" +
                                "[END_ARRAY]: prevToken = " + prevToken
+
                                ", curToken = " + token + ",
curFieldName = " +
                                curFieldName + ", stopField = " +
stopField +
                                " ... terminate with '}' and RETURN
string");
                        }
                        return strBldr.toString();
                    }

                } /* endif (prevToken isScalar && STOP-BY-FIELD) */

            } /* endif (prevToken isScalar) */

        } /* endif (prevToken != null) */

        /* Done with prevToken, process current input token next */

        if (token.isScalarValue()) { /* current token is SCALAR */

            strBldr.append("\"" + curFieldName + "\": " +
objectValue(parser));
            return jsonToString(
                        stopField, token, parser, strBldr,
fromObjectArray);
        }

        if (JsonToken.START_OBJECT == token) { /* input token is OBJECT
*/

            if (curFieldName != null) {
                strBldr.append("\"" + curFieldName + "\": {\n");
            } else {
                strBldr.append("{\n");
            }
            return jsonToString(
                        stopField, token, parser, strBldr,
fromObjectArray);

        } else if (JsonToken.START_ARRAY == token) { /* input Token is
ARRAY */

            if (debugAll || debugJsonToStringArray) {
                System.out.println(
                    "--- START_ARRAY --- prevToken = " + prevToken +
                    ", curToken = " + token + ", curFieldName = " +
                    curFieldName + ", stopField = " + stopField +
                    ", isScalar = " + token.isScalarValue() +
                    ", fromObjectArray = " + fromObjectArray +
                    " ... get NEXT TOKEN");
            }
```

```
            token = parser.nextToken();

            if (debugAll || debugJsonToStringArray) {
                System.out.println(
                    "--- START_ARRAY --- nextToken = " + token +
                    ", curFieldName = " + curFieldName + ", stopField =
" +
                    stopField + ", isScalar = " + token.isScalarValue()
+
                    ", fromObjectArray = " + fromObjectArray);
            }

            if (token == null) {
                System.out.println(
                    "*** WARNING: null next token after START_ARRAY. " +
                    "Invalid json document?");
                return strBldr.toString();
            }

            /* START_ARRAY then START_OBJECT: Handle ARRAY OF OBJECTS */

            if (JsonToken.START_OBJECT == token) {

                if (debugAll || debugJsonToStringArray) {
                    System.out.println("--- START_ARRAY then
START_OBJECT: " +
                                       "Handle ARRAY_OF_OBJECTS ---");
                }

                final StringBuilder tmpBldr = new StringBuilder();
                final String curArrayName = curFieldName;

                if (curArrayName != null) {
                    if (!curArrayName.equals(stopField)) {
                        tmpBldr.append("\"" + curFieldName + "\": [\n");
                    }
                }
                tmpBldr.append("{\n");


                while (token != null && JsonToken.END_ARRAY != token) {

                    /*
                     * When at the top of this loop, we know we're
handling an
                     * array of objects. We know that we're done with
that
                     * array of objects (has already been terminated
with ']')
                     * and so should terminate the object containing the
                     * array (with '}') if the following conditions are
met:
                     *
                     * 1. The previous token is a FIELD_NAME.
```

```
                     * 2. The current field name corresponding to the
current
                     *    token is the same as the name of the current
array.
                     * 3. The current token is END_OBJECT (meaning
we're at
                     *    the end of the object containing the array).
                     */
                    if (JsonToken.FIELD_NAME == prevToken &&
                        JsonToken.END_OBJECT == token &&
                        curFieldName != null &&
                        curFieldName.equals(curArrayName)) {

                        tmpBldr.append("}\n");

                        if (debugAll || debugJsonToStringArray) {
                            System.out.println(
                                "--- TOP ARRAY_OF_OBJECTS LOOP: " +
                                "array element END_OBJECT - " +
                                "terminate with '}' - prevToken = " +
                                prevToken + ", curToken = " + token +
                                ", curFieldName = " + curFieldName +
                                ", stopField = " + stopField +
                                " , fromObjectArray = " +
fromObjectArray +
                                " ... loop to continue or end of DOC");
                        }

                        String nextFieldName = curFieldName;
                        while(token != null) {

                            token = parser.nextToken();
                            nextFieldName = parser.getCurrentName();

                            if (debugAll || debugJsonToStringArray) {
                                System.out.println(
                                    "--- TOP ARRAY_OF_OBJECTS LOOP: " +
                                    "array termination inner loop - " +
                                    "nextToken = " + token +
                                    ", nextFieldName = " +
nextFieldName);
                            }

                            if (token.isScalarValue()) {

                                tmpBldr.append(",\n");
                                tmpBldr.append("\"" + nextFieldName +
                                               "\": " +
objectValue(parser));
                                break;
                            }

                            if (JsonToken.START_OBJECT == token) {

                                tmpBldr.append(",\n");
```

```
                                if (nextFieldName != null) {
                                    tmpBldr.append("\"" + nextFieldName
+
                                                       "\": { kkkk\n");
                                } else {
                                    tmpBldr.append("{ kkkk\n");
                                }
                                break;
                            }

                            if (JsonToken.START_ARRAY == token) {

                                tmpBldr.append(",\n");
                                if (nextFieldName != null) {
                                    tmpBldr.append("\"" + nextFieldName
+
                                                       "\": [ kkkk\n");
                                } else {
                                    tmpBldr.append("[ kkkk\n");
                                }
                                break;
                            }

                            if ((nextFieldName != null &&
                        nextFieldName.equals(stopField)) ||
                                (nextFieldName == null &&
                                 JsonToken.END_OBJECT == token)) {

                                final String finalRetStr =

strBldr.append(tmpBldr.toString()).toString();

                                if (debugAll || debugJsonToStringArray)
{
                                    System.out.println(
                                        "--- TOP ARRAY_OF_OBJECTS LOOP:
" +
                                        "DONE - RETURN FINAL STRING
=\n" +
                                        finalRetStr);
                                }
                                return finalRetStr;
                            }


                        } /* end loop */

                        /* More tokens to process. Recurse. */

                        if (debugAll || debugJsonToStringArray) {
                            System.out.println(
                                "--- TOP ARRAY_OF_OBJECTS LOOP: " +
                                "curToken = " + token + ", curFieldName
= " +
                                nextFieldName + ", stopField = " +
```

```
stopField +
                                    " , fromObjectArray = " +
fromObjectArray +
                                    "-  more tokens to process ...
RECURSE");
                            }
                            jsonToString(stopField, token, parser, tmpBldr,
false);

                        } else { /* Not end of outer object containing
objArray */

                            if (debugAll || debugJsonToStringArray) {
                                System.out.println(
                                    "--- TOP ARRAY_OF_OBJECTS LOOP: " +
                                    "prevToken = " + prevToken + ",
curToken = " +
                                    token + ", curFieldName = " +
                                    curFieldName + ", stopField = " +
stopField +
                                    " , fromObjectArray = " +
fromObjectArray +
                                    "-  NOT END OF ARRAY ... RECURSE");
                            }
                            jsonToString(stopField, token, parser, tmpBldr,
true);

                        } /* endif (FIELD_NAME then END_OBJECT && array
name) */


                        if (debugAll || debugJsonToStringArray) {
                            System.out.println(
                                "--- TOP ARRAY_OF_OBJECTS LOOP: " +
                                "EXIT jsonToString() - prevToken = " +
                                prevToken + ", curToken = " + token +
                                ", curFieldName = " + curFieldName +
                                ", stopField = " + stopField +
                                " , fromObjectArray = " + fromObjectArray +
                                " ... loop to continue or end of DOC");
                        }


                        if (JsonToken.FIELD_NAME == prevToken &&
                            JsonToken.FIELD_NAME == token &&
                            curFieldName != null) {

                            if (debugAll || debugJsonToStringArray) {
                                System.out.println(
                                    "--- IN ARRAY_OF_OBJECTS LOOP: " +
                                    "FIELD_NAME then FIELD_NAME - " +
                                    "curToken = " + token +
                                    ", curFieldName = " + curFieldName +
                                    " ... get NEXT TOKEN");
                            }
```

```
                        token = parser.nextToken();

                        if (debugAll || debugJsonToStringArray) {
                            System.out.println(
                                "--- IN ARRAY_OF_OBJECTS LOOP: " +
                                "FIELD_NAME then FIELD_NAME - " +
                                "nextToken = " + token + ",
nextFieldName = " +
                                parser.getCurrentName());
                        }

                        if (JsonToken.END_OBJECT == token) {

                            if (debugAll || debugJsonToStringArray) {
                                 System.out.println(
                                   "--- IN ARRAY_OF_OBJECTS LOOP: " +
                                   "FIELD_NAME then FIELD_NAME - " +
                                   "nextToken = END_OBJECT ... BREAK " +
                                   "out of loop ... subString =\n" +
                                   tmpBldr.toString());
                            }
                            break;

                        } else { /* nextToken NOT END_OBJECT */

                            final String finalRetStr = strBldr.append(
                                tmpBldr.toString()).toString();

                            if (debugAll || debugJsonToStringArray) {
                                 System.out.println(
                                   "--- IN ARRAY_OF_OBJECTS LOOP: " +
                                   "FIELD_NAME then FIELD_NAME - " +
                                   "nextToken = " + token +
                                   "(NOT END_OBJECT) ... DONE - " +
                                   "RETURN FINAL STRING = \n" +
finalRetStr);
                            }
                            return finalRetStr;

                        } /* endif (nextToken END_OBJECT or NOT) */

                    } else if (JsonToken.FIELD_NAME == prevToken &&
                               JsonToken.START_OBJECT == token &&
                               curFieldName != null) {

                        if (debugAll || debugJsonToStringArray) {
                             System.out.println(
                               "--- IN ARRAY_OF_OBJECTS LOOP: " +
                               "FIELD_NAME then START_OBJECT - " +
                               "curToken = " + token + ", curFieldName
= " +
                               parser.getCurrentName());
                        }
```

```
                        token = parser.nextToken();
                        final String nextFieldName =
parser.getCurrentName();

                        if (JsonToken.FIELD_NAME == token &&
                            nextFieldName != null) {
                            tmpBldr.append(",\n");

                        } else if (JsonToken.END_OBJECT == token &&
                                   nextFieldName != null) {
                            tmpBldr.append("}\n");
                        }

                        if (debugAll || debugJsonToStringArray) {
                             System.out.println(
                               "--- IN ARRAY_OF_OBJECTS LOOP: " +
                               "FIELD_NAME then START_OBJECT - " +
                               "nextToken = " + token + ",
nextFieldName = " +
                               nextFieldName + " ... current subString
=\n" +
                               tmpBldr.toString());
                        }

                    } else {

                        token = parser.nextToken();

                        if (debugAll || debugJsonToStringArray) {
                             System.out.println(
                               "--- IN ARRAY_OF_OBJECTS LOOP: " +
                               "*** ELSE *** nextToken = " + token +
                               ", nextFieldName = " +
                               parser.getCurrentName() +
                               " ... current subString =\n" +
                               tmpBldr.toString());
                        }

                    }/* endif FIELD_NAME && FIELD_NAME && fieldName !=
null */

                    if (debugAll || debugJsonToStringArray) {
                         System.out.println(
                           "--- IN ARRAY_OF_OBJECTS LOOP: " +
                           "END OF LOOP - CONTINUE TO TOP OF LOOP" );
                    }

                } /* end ARRAY_OF_OBJECTS loop */

                if (debugAll || debugJsonToStringArray) {
                     System.out.println(
                       "--- OUT ARRAY_OF_OBJECTS LOOP:  " +
                       "current subString = \n" + tmpBldr.toString());
                }
```

```
                final String tmpStr = tmpBldr.toString();
                final String retStr = strBldr.append(tmpStr).toString();

                if (debugAll || debugJsonToStringArray) {
                     System.out.println(
                       "--- OUT ARRAY_OF_OBJECTS LOOP:  " +
                       "ENTER jsonToString() - BEGIN FINAL RETURN
string ...");
                }

                final String finalRetStr =
                    jsonToString(stopField, token, parser, strBldr,
false);

                if (debugAll || debugJsonToStringArray) {
                     System.out.println(
                       "--- OUT ARRAY_OF_OBJECTS LOOP:  EXIT " +
                       "jsonToString() - RETURN FINAL RETURN string =
\n" +
                       finalRetStr);
                }
                return finalRetStr;

            } /* endif START_OBJECT after START_ARRAY & ARRAY OF
OBJECTS */


            /* -- START_ARRAY then NOT START_OBJECT: ARRAY OF SCALARS
-- */

            if (debugAll || debugJsonToStringArray) {
                System.out.println("--- START_ARRAY then SCALAR: " +
                                   "Enter ARRAY_OF_SCALARS loop ---");
            }

            strBldr.append("\"" + curFieldName + "\": [\n");
            while (token != null && JsonToken.END_ARRAY != token) {
                strBldr.append(objectValue(parser));
                token = parser.nextToken();
                if (JsonToken.END_ARRAY == token) {
                    strBldr.append("\n]");
                } else {
                    strBldr.append(",\n");
                }
            }/* end loop: ARRAY_OF_SCALARS */

            return jsonToString(stopField, token, parser, strBldr,
false);

        } else if(JsonToken.END_OBJECT == token) {

            strBldr.append("\n}");
            return jsonToString(
                       stopField, token, parser, strBldr,
fromObjectArray);
```

```
        } else  { /* DEFAULT: all other values of current input token */

            return jsonToString(
                        stopField, token, parser, strBldr,
fromObjectArray);

        } /* endif (START_OBJECT else START_ARRAY else END_OBJECT) */
    }

    private Object objectValue(JsonParser parser) throws IOException {
        final JsonToken currentToken = parser.getCurrentToken();
        if (currentToken == JsonToken.VALUE_STRING) {
            return "\"" + parser.getText() + "\"";
        } else if (currentToken == JsonToken.VALUE_NUMBER_INT ||
                currentToken == JsonToken.VALUE_NUMBER_FLOAT) {
            return parser.getNumberValue();
        } else if (currentToken == JsonToken.VALUE_TRUE) {
            return Boolean.TRUE;
        } else if (currentToken == JsonToken.VALUE_FALSE) {
            return Boolean.FALSE;
        } else if (currentToken == JsonToken.VALUE_NULL) {
            return null;
        } else {
            return "\"" + parser.getText() + "\"";
        }
    }

    private int getIntFieldValue(final String objectName,
                                 final String fieldName,
                                 final JsonParser parser) throws
IOException {
        int nObjects = 0;
        JsonToken token = parser.nextToken();
        while (token != null) {
            String curFieldName = parser.getCurrentName();
            if (objectName.equals(curFieldName)) {
                token = parser.nextToken();
                while (token != null) {
                    curFieldName = parser.getCurrentName();
                    if (fieldName.equals(curFieldName)) {
                        token = parser.nextToken();
                        if (token != JsonToken.VALUE_NUMBER_INT) {
                            System.out.println("getIntFieldValue:
WARNING - " +
                                "for object " + objectName + ", value
of " +
                                "field " + fieldName + " is NOT an
integer [" +
                                parser.getText() + "]");
                            return nObjects;
                        }
                        nObjects = parser.getNumberValue().intValue();
                        return nObjects;
                    }
```

```
                    token = parser.nextToken();
                }
            }
            token = parser.nextToken();
        }
        System.out.println("getIntFieldValue: WARNING - could not find
" +
                           "field in given object [object=" +
objectName +
                           ", field=" + fieldName + "]");
        return nObjects;
    }
}
```

Secure Elasticsearch using Sheild

The following are the steps to take to configure an Elasticsearch cluster to run securely. The descriptions provided in each sub-section below are based on the following list of assumptions and requirements:

Assumptions about the Secure Elasticsearch Cluster

- The 2.4.6 version of the Elasticsearch distribution is installed under the directory `/opt/es/elasticsearch`.
- The 2.4.6 version of the Shield adapter (and license) is installed in the Elasticsearch configuration (see below).
- There are three nodes hosting the Elasticsearch cluster, named `eshost1`, `eshost2`, and `eshost3` respectively, each having network connectivity with the other nodes of the Elasticsearch cluster, as well as the nodes of the Oracle NoSQL store.
- The value used when specifying the `node.name` property for each node of the Elasticsearch cluster is the `hostname` of the corresponding Elasticsearch node.
- The cluster that is deployed is named `escluster` (`cluster.name`).
- The port used for node-to-node communication within the cluster itself is `29000` (`transport.tcp.port`).
- The port used by clients of the cluster when communicating over HTTPS with any node in the cluster (for example, to send Full Text Search queries), is `29100` (`http.port`).
- For simplicity, all passwords are set to `No_Sql_00`.
- The nodes of the Elasticsearch cluster each generate a public/private keypair with an alias that is unique relative to the aliases of the keypairs generated by the other nodes in the cluster. This is a requirement because the public certificate from each Elasticsearch node will be installed in the truststore of the other nodes of the cluster, as well as the truststore of each node in the Oracle NoSQL store. This is necessary not only for secure communication between the Oracle NoSQL store and the Elasticsearch cluster, but also for secure communication between the nodes of the cluster itself. To achieve the required alias uniqueness, each alias will include the hostname of the Elasticsearch node that generates the keypair.

- Although the alias of the keypair generated by each node must be unique, all of those keypairs share the same Distinguished Name (DN); with Common Name (CN) equal to `esuser`.

> **Note:**
>
> The Shield security plugin used by Elasticsearch employs Public Key Infrastructure (PKI) for user authentication. As a result, when a node in the NoSQL store attempts to communicate with an Elasticserch node, the Elasticsearch node presents a certificate to the store node, which the store node must trust in order for the communication to succeed. There are two options for establishing PKI certificate trust:
>
> - Self-signed public certificates
> - Public certificates signed by a Certificate Authority (CA)
>
> The secure Elasticsearch cluster presented here uses self-signed certificates. As described above, using this option, each node in the Elasticsearch cluster must provide its own certificate with unique alias; and each such certificate must be installed in the truststore of any service (example: the Oracle NoSQL store) or client that wishes to communicate with the Elasticsearch cluster.
>
> Although obtaining and installing a single CA-signed certificate is less cumbersome than installing a self-signed certificate from each of the Elasticsearch nodes, the use of self-signed certificates can be more instructive with respect to PKI concepts. Once you understand how to work with self-signed certificates, changing your deployment to employ the CA-signed option should be straightforward.

Install Elasticsearch and the Shield Plugin

You can find the 2.4.6 version of Elasticsearch, Shield, and the Shield license at the following URLs:

- https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/tar/elasticsearch/2.4.6/elasticsearch-2.4.6.tar.gz
- https://download.elastic.co/elasticsearch/release/org/elasticsearch/plugin/shield/2.4.6/shield-2.4.6.zip
- https://download.elastic.co/elasticsearch/release/org/elasticsearch/plugin/license/2.4.6/license-2.4.6.zip

On each Elasticsearch node (`eshost1`, `eshost2`, and `eshost3`), create the directory `/opt/es`, download `elasticsearch-2.4.6.tar.gz` to that directory, and install the Elasticsearch software. For example, on each host do the following:

```
mkdir –p /opt/es/install-xfer/certs
```

Use `curl`, `wget` or your browser to download `elasticsearch-2.4.6.tar.gz` to `/opt/es`, then

```
cd /opt/es
tar xzvf elasticsearch-2.4.6.tar.gz
ln –s elasticsearch-2.4.6 elasticsearch
```

Download the Shield distribution and its corresponding license to a temporary directory on each node (example: `/tmp`). Do not place those zip files under the `/opt/es/elasticsearch` home directory; otherwise installation errors can occur. Once you have downloaded the Shield distributions and its corresponding license, install Shield by doing the following:

```
export JAVA_HOME=/opt/java/java8  [if necessary]

cd /opt/es
bin/plugin install –v file:///tmp/elasticsearch-shield-license-2.4.6.zip
bin/plugin install –v file:///tmp/elasticsearch-shield-2.4.6.zip
```

> **Note:**
>
> Java 8 or greater is required to install the Shield plugin, as well as to deploy the Elasticsearch cluster itself. Thus, if the default version of Java on your Elasticsearch nodes is less than Java 8, then you should install Java 8 or greater on each node, and set the `JAVA_HOME` environment variable to point to that installation before installing the Shield plugin or deploying the cluster.
>
> Also, when you initially install Shield, a 30 day trial license is installed that allows access to all Shield features. Although the 30 day trial license should suffice to run this example, you can purchase a subscription at the end of the trial period if you want to keep using the full functionality of Shield; otherwise, you can use Shield in a degraded mode after expiration, where the monitoring feature is disabled.

Create and Install a Public/Private Keypair in the Elasticsearch Keystore

On each Elasticsearch node, generate a public/private keypair that clients of Elasticsearch can use to execute secure queries on the data indexed in the cluster. For example, on `eshost1` execute:

```
keytool –genkeypair
    –alias elasticsearch-eshost1
    –keystore /opt/es/elasticsearch/config/shield/elasticsearch.keys
    -keyalg RSA
    –keysize 2048
    –validity 1712
    –storepass No_Sql_00
    –keypass No_Sql_00
    -dname CN=esuser,OU=es.org,L=es.city,S=es.state,C=US
    -ext san=dns:localhost,dns:eshost1,dns:eshost2,dns:eshost3,
            dns:kvhost1,dns:kvhost2,dns:kvhost3
```

This command will generate a keypair with alias `elasticsearch-eshost1`. It will place the keypair in that node's keystore (`elasticsearch.keys`) if the keystore already exists; otherwise it will create the keystore before generating the keypair.

Export the Public Certificate and Install it in the Truststore

On each Elasticsearch node, export the public certificate from the keypair generated above. Store the resulting certificate file in a directory outside of the Shield config directory; for example, `/opt/es/install-xfer/certs`. This will facilitate distribution of the certificate to the other nodes in the cluster as well as clients of Elasticsearch (example: the Oracle NoSql store). For example, on `eshost1` execute:

```
keytool –export
    –alias elasticsearch-eshost1
    –keystore /opt/es/elasticsearch/config/shield/elasticsearch.keys
    -storepass No_Sql_00
    -file /opt/es/install-xfer/certs/elasticsearch-eshost1.crt
```

This command will retrieve the public certificate with the given alias from the keystore and place it in the certificate file (`elasticsearch-eshost1.crt`) located in the separate transfer directory (`install-xfer/certs`).

Once the certificate file is available, import (install) the public certificate into the node's truststore. For example, on `eshost1` execute:

```
keytool –importcert
    –alias elasticsearch-eshost1
    -file /opt/es/install-xfer/certs/elasticsearch-eshost1.crt
    –keystore /opt/es/elasticsearch/config/shield/elasticsearch.trust
    -storepass No_Sql_00
    -keypass No_Sql_00
    –noprompt
```

If the node's truststore already exists, this command will install the public certificate from the specified file into that truststore (`elasticsearch.trust`); otherwise the truststore is created prior to importing the certificate.

Convert the Public/Private Keys to OpenSSL Format (pem/key)

On each Elasticsearch node, retrieve the previously generated public/private keypair from the node's keystore as a PKCS12 file. For example, on `eshost1`:

```
keytool –importkeystore
    -srckeystore /opt/es/install-xfer/certs/elasticsearch.keys
    –srcalias elasticsearch-eshost1
    -srcstorepass No_Sql_00
    –dstkeystore /opt/es/install-xfer/certs/elasticsearch-eshost1.p12
    -deststoretype PKCS12
    -deststorepass No_Sql_00
    -destkeypass No_Sql_00
```

Next, retrieve the public certificate – in PEM format – from the PKCS12 file that was just retrieved. For example:

```
openssl pkcs12
    -in /opt/es/install-xfer/certs/elasticsearch-eshost1.p12
    -passin pass:No_Sql_00
    –out /opt/es/install-xfer/certs/elasticsearch-eshost1.pem
    -nokeys
```

Finally, retrieve the private key file from that PKCS12 file. For example:

```
openssl pkcs12
    -in /opt/es/install-xfer/certs/elasticsearch-eshost1.p12
    -passin pass:No_Sql_00
    –out /opt/es/install-xfer/certs/elasticsearch-eshost1.pkey
    -nocerts
```

The commands above produce two files, `elasticsearch-eshost1.pem` and `elasticsearch-eshost1.pkey` that can be installed on clients of Elasticsearch and used to execute secure queries against data indexed by the cluster. In the initial stages of cluster configuration, these files can be used to verify that Elasticsearch security has been configured correctly.

Modify the Elasticsearch and Shield Configuration Files

To complete the configuration of Elasticsearch to run securely using the Shield plugin, the following YAML configuration files must be modified on each Elasticsearch node:

- `/opt/es/elasticsearch/config/elasticsearch.yml`
- `/opt/es/elasticsearch/config/shield/role_mapping.yml`

On each Elasticsearch node, edit the files listed above and make the following modifications.

1. Add the following lines to `elasticsearch.yml`

```
shield:
  enabled: true
  authc:
    realms:
        pki1:
            type: pki
            enabled: true
            order: 0
  transport:
      ssl: true
      ssl.client.auth: required
  http:
      ssl: true
      ssl.client.auth: required
  ssl:
      keystore:"current": true,
          path: /opt/es/elasticsearch/config/shield/
elasticsearch.keys
```

```
            password: No Sql_00
            key_password: No_Sql_00
        truststore:
            path: /opt/es/elasticsearch/config/shield/
elasticsearch.trust
            password: No Sql_00
```

2. Add the following three lines to `role_mapping.yml`

```
admin:
    - "CN=esuser,OU=es.org.unit,O=es.org,L=es.city,ST=es.state,C=US"
    -
"CN=FTS,OU=nosql.org.unit,O=nosql.org,L=nosql.city,ST=nosql.state,C=
US"
```

Without these additions to the Elasticsearch and Shield configurations, any attempt by you or the Oracle NoSQL store to communicate with the secure Elasticsearch cluster will encounter errors related to either authentication or TLS/SSL failures.

At this point, there is still more to do to configure the Elasticsearch cluster for secure communication with an Oracle NoSQL store. But before that can be done, you must first configure and deploy the store itself. If you are confident that your current Elasticsearch security configuration is correct, then you can go directly to Deploying and Configuring a Secure Oracle NoSQL Store to deploy the secure Oracle NoSQL store and configure it for secure communication with the Elasticsearch cluster. But if you prefer to verify that what you have done so far is correct, then execute the steps presented in the next sub-section.

[Optional] Verify Elasticsearch Security is Configured Correctly

Before moving on to deploying and configuring a secure Oracle NoSQL store, you may wish to verify that queries can indeed be successfully (and securely) executed against the Elasticsearch cluster with its current configuration. To do this, you must first install each Elasticsearch node's PEM formatted public certificate and private key on any client from which a query will be sent to Elasticsearch.

For example, suppose your client node is named `clhost1`. And suppose you copy the public/private PEM files from each Elasticsearch node to the `/tmp` directory of `clhost1`. That is, on `clhost1`,

```
scp <username>@eshost1:/opt/es/install-xfer/certs/elasticsearch-
eshost1.pem  /tmp
scp <username>@eshost1:/opt/es/install-xfer/certs/elasticsearch-
eshost1.pkey /tmp

scp <username>@eshost2:/opt/es/install-xfer/certs/elasticsearch-
eshost2.pem  /tmp
scp <username>@eshost2:/opt/es/install-xfer/certs/elasticsearch-
eshost2.pkey /tmp

scp <username>@eshost3:/opt/es/install-xfer/certs/elasticsearch-
eshost3.pem  /tmp
scp <username>@eshost3:/opt/es/install-xfer/certs/elasticsearch-
eshost3.pkey /tmp

ls /tmp
```

```
 elasticsearch-eshost1.pem
 elasticsearch-eshost1.pkey

 elasticsearch-eshost2.pem
 elasticsearch-eshost2.pkey

 elasticsearch-eshost3.pem
 elasticsearch-eshost3.pkey
```

Next, deploy the secure Elasticsearch cluster by logging in to each Elasticsearch node and executing the following commands:

On `eshost1`

```
cd /scratch/es
export JAVA_HOME=/opt/java/java8  [if necessary]

./elasticsearch/bin/elasticsearch
    --cluster.name escluster
    --node.name eshost1
    --transport.tcp.port 29000
    --http.port 29100
    --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

On `eshost2`

```
cd /scratch/es
export JAVA_HOME=/opt/java/java8  [if necessary]

./elasticsearch/bin/elasticsearch
    --cluster.name escluster
    --node.name eshost2
    --transport.tcp.port 29000
    --http.port 29100
    --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

On `eshost3`

```
cd /scratch/es
export JAVA_HOME=/opt/java/java8  [if necessary]

./elasticsearch/bin/elasticsearch
    --cluster.name escluster
    --node.name eshost3
    --transport.tcp.port 29000
    --http.port 29100
    --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

Once the Elasticsearch cluster has been deployed, you can send queries from the client node to any of the nodes making up the Elasticsearch cluster. For example,

```
curl –k –E /tmp/elasticsearch-eshost1.pem
    –-key /tmp/elasticsearch-eshost1.pkey
    -X GET 'https://eshost1:29100/_cat/nodes'

curl –k –E /tmp/elasticsearch-eshost2.pem
    –-key /tmp/elasticsearch-eshost2.pkey
    -X PUT 'https://eshost2:29100/indices'

curl –k –E /tmp/elasticsearch-eshost3.pem
    –-key /tmp/elasticsearch-eshost3.pkey
    -X GET 'https://eshost3:29100/_cat/indices'
```

Be sure to use the public certificate and private key corresponding to the node to which you send the query.

After verifying that the Elasticsearch cluster is configured correctly and can execute secure queries, `shutdown/kill [crtl-c]` the Elasticsearch process on each node.

At this point, we are ready to deploy a secure Oracle NoSQL store and configure it for communication with the secure Elasticsearch cluster from this section. See Deploying and Configuring a Secure Oracle NoSQL Store.

Deploying and Configuring a Secure Oracle NoSQL Store

There are a number of different methods to deploy and configure an Oracle NoSQL store for secure access. This section presents one particular set of steps you can take to deploy and configure such a store. For other methods, see Security Configuration in the *Security Guide*.

Additionally, since the store that is deployed must communicate with the secure Elasticsearch cluster from Secure Elasticsearch using Sheild, this section also shows how to generate and install the private keys and public certificates needed by the store and cluster for secure communication.

Whether you prefer the method presented here or one of the other methods presented in the *Security Guide*, the following assumptions and requirements apply when configuring an Oracle NoSQL store for secure deployment and communication with a secure Elasticsearch cluster:

Assumptions about the Secure Oracle NoSQL Store

- The Oracle NoSQL Database distribution is installed under the directory `/opt/ondb/kv`.
- There are three nodes hosting the store, named `kvhost1`, `kvhost2`, and `kvhost3` respectively.
- The store is deployed with a replication factor (rf) of `3`, and is named `mystore`.
- An admin service, listening on port `5000`, is deployed on each of the store's nodes.

- The range of ports used to support high availability (harange) consists of port `5002` through `5007`.
- One storage node (SN) per store host will be deployed (capacity 1), with default values for the number of cpu's and memory (num_cpus 0 and memory_mb 0).
- The contents of the shards (replication groups) managed by the store are located under the storage directory `/disk1/shard` on each node of the store; where the size specified for each storage directory is `1GB` (1,000,000,000 bytes).
- For convenience, the password manager the store uses to store and retrieve passwords for access to the store's keystore and truststore is a password file (available in all editions of Oracle NoSQL Database), rather than the Oracle Wallet (available in only the Enterprise Edition).
- For simplicity, all passwords are set to `No_Sql_00`.
- The name of the alias used in the public/private keypair generated by the store and provided to Elasticsearch for secure communication with the store, is `FTS`. Note that this is a requirement, as communication with a secure store will fail if Elasticsearch responds to a request from the store by presenting a certificate with an alias different than `FTS`.
- A user with administrative privileges is provisioned in the store's access control list. The name given to this user is `FTS`; the same as the alias of the keypair the store generates for Elasticsearch. Although the user name is not required to be the same as the alias, it is given that value for consistency, and to avoid confusion.

Provision the Store Boot Node for Secure Deployment and Elasticsearch Communication

All of the commands presented in this sub-section are executed on only the first (boot) node of the store (example: `kvhost1`). Using the assumptions previously listed, when provisioning the boot node of a store that will be deployed with security, the first command to execute is:

On `kvhost1`

```
export JAVA_HOME=/opt/java/java8  [if necessary]

java -jar /opt/ondb/kv/lib/kvstore.jar makebootconfig
  -root /opt/ondb/kvroot
  -config config.xml
  -port 5000
  -host kvhost1
  -harange 5002,5007
  -capacity 1
  -num_cpus 0
  -memory_mb 0
  -storagedir /disk1/shard
  -storagedirsize 1000000000
  -store-security configure
  -pwdmgr pwdfile
  -kspwd No_Sql_00
```

The command above creates the security directory `/opt/ondb/kvroot/security` on the store's boot node `kvhost1`, and populates it with security artifacts such as the store's keystore (`store.keys`) and trustore (`store.trust`). For convenience, it

also creates artifacts that can be distributed to clients for secure access to the store (`client.trust` and `client.security`). After executing the command above, you should see the following files in the security directory:

```
ls /opt/ondb/kvroot/security
  store.trust
  client.trust
  client.security
  security.xml
  store.keys
  store.passwd
```

Athough the command above is necessary to deploy a secure store, it is not sufficient for secure communication with the Elasticsearch cluster from Secure Elasticsearch using Sheild. To facilitate secure communication with Elasticsearch, a public/private keypair with the alias FTS must be generated and installed in the store's keystore. For example,

On `kvhost1`

```
keytool -genkeypair
  -alias FTS
  -keystore /opt/ondb/kvroot/security/store.keys
  -keyalg RSA
  -keysize 2048
  -validity 1712
  -storepass No_Sql_00
  -keypass No_Sql_00
  -dname CN=FTS,OU=nosql.org,L=nosql.city,S=nosql.state,C=US
  -ext san=dns:localhost,dns:eshost1,dns:eshost2,dns:eshost3,
          dns:kvhost1,dns:kvhost2,dns:kvhost3
```

After generating the keypair above, the public certificate from that keypair must be exported from the keystore. In order for any node of the Elasticsearch cluster to securely communicate with the NoSQL store, the Elasticsearch node must send this certificate to the store. Thus, the certificate produced by the following export command will ultimately be installed on each node of the Elasticsearch cluster. See Install the Full Text Search Public Certificate in Elasticsearch. On `kvhost1`,

```
keytool -export
  -alias FTS
  -keystore /opt/ondb/kvroot/security/store.keys
  -storepass No_Sql_00
  -file /opt/ondb/kvroot/security/FTS.crt
```

Whereas the FTS public certificate created by this command must be presented to the Oracle NoSQL store by each Elasticsearch node when the node attempts to communicate with the store, the store must also present the Elasticsearch node's public certificate. This is because the model for secure communication between Elasticsearch and Oracle NoSQL requires mutual authentication. As a result, the

public certificates created on each of the Elasticsearch nodes in Secure Elasticsearch using Sheild must be retrieved and installed in the store's truststore. For example,

```
scp <username>@eshost1:/opt/es/install-xfer/certs/elasticsearch-
eshost1.crt /opt/ondb/kvroot/security
scp <username>@eshost2:/opt/es/install-xfer/certs/elasticsearch-
eshost2.crt /opt/ondb/kvroot/security
scp <username>@eshost3:/opt/es/install-xfer/certs/elasticsearch-
eshost3.crt /opt/ondb/kvroot/security

keytool -importcert
  -alias elasticsearch-eshost1
  -file /opt/ondb/kvroot/security/elasticsearch-eshost1.crt
  -keystore /opt/ondb/kvroot/security/store.trust
  -storepass No_Sql_00
  -keypass No_Sql_00
  -noprompt

keytool -importcert
  -alias elasticsearch-eshost2
  -file /opt/ondb/kvroot/security/elasticsearch-eshost2.crt
  -keystore /opt/ondb/kvroot/security/store.trust
  -storepass No_Sql_00
  -keypass No_Sql_00
  -noprompt

keytool -importcert
  -alias elasticsearch-eshost3
  -file /opt/ondb/kvroot/security/elasticsearch-eshost3.crt
  -keystore /opt/ondb/kvroot/security/store.trust
  -storepass No_Sql_00
  -keypass No_Sql_00
  -noprompt
```

At this point, the store's boot node is configured for secure deployment, and its security directory has been provisioned with the necessary security artifacts for communication with the Elasticsearch cluster from Secure Elasticsearch using Sheild.

The final step in the provisioning process is to install the same security artifacts created on the boot node in each of the remaining nodes of the store. This is accomplished by simply copying the boot node's security directory to each of those other nodes. For example, if the boot node is `kvhost1`, then you would do something like the following from that node:

```
scp -r /opt/ondb/kvroot/security <username>@kvhost2:/opt/ondb/kvroot
scp -r /opt/ondb/kvroot/security <username>@kvhost3:/opt/ondb/kvroot
```

Configure the Store's Remaining non-Boot Nodes for Security

Once the store's boot node is configured for security and the security directory of all of the nodes in the store have been fully provisioned as described in the previous sub-section, the remaining (non-boot) nodes of the store must also be configured for security. This is accomplished by using Java 8 or greater to execute, respectively, the following commands on each of the remaining nodes.

On `kvhost2`

```
java -jar /opt/ondb/kv/lib/kvstore.jar makebootconfig
  -root /opt/ondb/kvroot
  -config config.xml
  -port 5000
  -host kvhost2
  -harange 5002,5007
  -capacity 1
  -num_cpus 0
  -memory_mb 0
  -storagedir /disk1/shard
  -storagedirsize 1000000000
  -store-security enable
  -pwdmgr pwdfile
```

On `kvhost3`

```
java -jar /opt/ondb/kv/lib/kvstore.jar makebootconfig
  -root /opt/ondb/kvroot
  -config config.xml
  -port 5000
  -host kvhost3
  -harange 5002,5007
  -capacity 1
  -num_cpus 0
  -memory_mb 0
  -storagedir /disk1/shard
  -storagedirsize 1000000000
  -store-security enable
  -pwdmgr pwdfile
```

At this point, the store is configured and fully provisioned for secure deployment. The following sub-sections describe how this is accomplished.

Start Each Node of the NoSQL Store

Using Java 8 or greater, execute the following command on each node of the store.

On `kvhost1`, `kvhost2`, and `kvhost3`

```
java -jar /opt/ondb/kv/lib/kvstore.jar start
  -root /opt/ondb/kvroot
  -config config.xml
```

Deploy the Secure NoSQL Store

To deploy an Oracle NoSQL store based on the assumptions listed previously, first create a text file containing the following Oracle NoSQL administrative commands that can be executed as a script from the Oracle NoSQL Admin CLI.

```
configure -name mystore
plan deploy-zone -name zn1 -rf 3 -wait
```

```
plan deploy-sn -znname zn1 -host kvhost1 -port 5000 -wait
plan deploy-admin -sn 1 -wait
pool create -name snpool
pool join -name snpool -sn sn1

plan deploy-sn -znname zn1 -host kvhost2 -port 5000 -wait
plan deploy-admin -sn 2 -wait
pool join -name snpool -sn sn2

plan deploy-sn -znname zn1 -host kvhost3 -port 5000 -wait
plan deploy-admin -sn 3 -wait
pool join -name snpool -sn sn3

change-policy -params "loggingConfigProps=oracle.kv.level=INFO;"

topology create -name snlayout -pool snpool -partitions 300
plan deploy-topology -name snlayout -plan sndeploy -wait

execute "CREATE USER root IDENTIFIED BY 'No_Sql_00' ADMIN";
```

Note that a user named root with `ADMIN` privileges will be created when the store is deployed. That user will be used to add other users to the store's access control list (ACL); for example, the user named `FTS` described previously.

Once you have created the command file above, start the Admin CLI and deploy the store by loading that file. For example, suppose the commands are stored in the file, `/tmp/deploy-secure-store.cmds`. You would then deploy the store by doing the following:

On `kvhost1`

```
java -jar /opt/ondb/kv/lib/kvstore.jar runadmin
  -host kvhost1
  -port 5000
  -security /opt/ondb/kvroot/security/client.security

  Logged in admin as anonymous
  Connected to Admin in read-only mode

kv-> load -file /tmp/deploy-secure-store.cmds

  Connected to Admin in read-only mode
  Store configured: mystore
  ....
  Created: snlayout
  Executed plan 13, waiting for completion...
  Plan 13 ended successfully
  Statement completed successfully

kv-> exit
```

> **Note:**
>
> The clocks on `kvhost1`, `kvhost2`, and `kvhost3` must by synchronized (by default, within a 2 second delta), otherwise store deployment will fail. To determine whether a failed deployment was caused by unsynchronized clocks, check the admin logs on the affected node `/opt/ondb/kvroot/mystore/log/adminN_0.log`.

Provision the `root` User

To provison the user named root that was created during store deployment, do the following:

On `kvhost1`

```
java -jar /opt/ondb/kv/lib/kvstore.jar securityconfig pwdfile create
  -file /opt/ondb/kvroot/security/root.passwd

java -jar /opt/ondb/kv/lib/kvstore.jar securityconfig pwdfile secret
  -file /opt/ondb/kvroot/security/root.passwd -set -alias root

  Enter the secret value to store: No_Sql_00
  Re-enter the secret value for verification: No_Sql_00
```

Create a properties file that you can use to access (login to) the Admin CLI as the root user just created. This file should contain the same entries as the default client.security file generated when the store was initially provisioned for security, along with entries that specify the username and password file specific to the root user. For example,

```
cp /opt/ondb/kvroot/security/client.security /opt/ondb/kvroot/security/
root.login

echo oracle.kv.auth.username=root >> /opt/ondb/kvroot/security/
root.login
echo oracle.kv.auth.pwdfile.file=
/opt/ondb/kvroot/security/root.passwd >> /opt/ondb/kvroot/security/
root.login
```

Create and Provision the FTS User For Indexing Data in Secure Elasticsearch

In a production system, you would not typically use the root user to create and populate tables and Secondary Indexes in the Oracle NoSQL store or Text Indexes in the Elasticsearch cluster. Instead, you would generally use the root user to create other client users of the store whose roles are specific to a particular task; for example, indexing data in Elasticsearch.

For this example, a user named FTS is created and granted the privileges needed to create and populate a table, as well as index the table's data in Elasticsearch. To do this, you need to first create an Admin CLI command file that contains entries such as:

```
execute 'CREATE ROLE ftsadmin'
execute 'GRANT SYSDBA TO ftsadmin'
```

```
execute 'GRANT READ_ANY TO ftsadmin'
execute 'GRANT WRITE_ANY TO ftsadmin'
execute 'CREATE USER FTS IDENTIFIED BY "No_Sql_00"'
execute 'GRANT ftsadmin TO USER FTS'
execute 'GRANT SYSADMIN TO USER FTS'
```

Then, assuming `/tmp/create-user-FTS.cmds` is the path to that command file, you create the user by logging into the Admin CLI as the `root` user and then loading the command file. For example,

On `kvhost1`

```
java -jar /opt/ondb/kv/lib/kvcli.jar runadmin
  -host kvhost1
  -port 5000
  -store mystore
  -security /opt/ondb/kvroot/security/root.login

  Logged in admin as root

kv-> load -file /tmp/create-user-FTS.cmds

  Statement completed successfully
  Statement completed successfully
  ....

kv-> exit
```

To complete the provisioning of the FTS user just created, you should create a password file for that user and install it in a directory (for example, `/tmp`) on the client node you will be using to load and index data. For completeness (and convenience), in a fashion similar to what was done for the `root` user, you should also create a properties file that can be used to login to the Admin CLI as the user FTS. For example,

On `kvhost1`

```
java -jar /opt/ondb/kv/lib/kvstore.jar
  securityconfig pwdfile create
  -file /tmp/FTS.passwd

java -jar /opt/ondb/kv/lib/kvstore.jar
  securityconfig pwdfile secret
  -file /tmp/FTS.passwd
  -set
  -alias FTS

  Enter the secret value to store: No_Sql_00
  Re-enter the secret value for verification: No_Sql_00

cp /opt/ondb/kvroot/security/client.security /tmp/FTS-client.login

echo oracle.kv.auth.username=FTS >> /opt/ondb/kvroot/security/FTS-
client.login
```

```
echo oracle.kv.auth.pwdfile.file=/tmp/FTS.passwd >> /tmp/FTS-
client.login

cp /opt/ondb/kvroot/security/client.trust /tmp
```

Once the three artifacts above (`FTS-client.login`, `FTS.passwd`, and `client.trust`) have been created and installed in the `/tmp` directory on `kvhost1`, you can install them on any client. For example,

```
scp /tmp/FTS-client.login <username>@clhost1:/tmp
scp /tmp/FTS.passwd <username>@clhost1:/tmp
scp /tmp/client.trust <username>@clhost1:/tmp
```

At this point the store is fully deployed and ready to interact with the Elasticsearch cluster.

The only thing left to do before running the example is to install the Oracle NoSQL store's public certificate (`alias=FTS`) in the truststore on each Elasticsearch node. See Install the Full Text Search Public Certificate in Elasticsearch.

Install the Full Text Search Public Certificate in Elasticsearch

The final set of steps that must be executed to complete the deployment of the system consisting of a secure Oracle NoSQL store and a secure Elasticsearch cluster is to retrieve the Oracle NoSQL store's public certificate with alias FTS, and install that certificate in the truststore of each Elasticsearch node. For example,

On `eshost1`, `eshost2`, and `eshost3`,

```
scp <username>@kvhost1:/opt/ondb/kvroot/security/FTS.crt /opt/es/
install-xfer/certs
keytool –importcert
  –alias FTS
  -file /opt/es/install-xfer/certs/FTS.crt
  –keystore /opt/es/elasticsearch/config/shield/elasticsearch.trust
  -storepass No_Sql_00
  –keypass No_Sql_00
  –noprompt
```

Once the store's FTS public certificate is installed on each Elasticsearch node, you can deploy the Elasticsearch cluster; which should now be able to communicate with the secure Oracle NoSQL store deployed in Deploying and Configuring a Secure Oracle NoSQL Store. For example, using Java 8 or greater,

On `eshost1`

```
cd /scratch/es

./elasticsearch/bin/elasticsearch
  --cluster.name escluster
  --node.name eshost1
  --transport.tcp.port 29000
  --http.port 29100
```

```
  --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

On `eshost2`

```
cd /scratch/es

./elasticsearch/bin/elasticsearch
  --cluster.name escluster
  --node.name eshost2
  --transport.tcp.port 29000
  --http.port 29100
  --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

On `eshost3`

```
cd /scratch/es

./elasticsearch/bin/elasticsearch
  --cluster.name escluster
  --node.name eshost3
  --transport.tcp.port 29000
  --http.port 29100
  --discovery.zen.ping.unicast.hosts
eshost1:29000,eshost2:29000,eshost3:29000
```

At this point you should now be able to do the following:

1. Execute the example program in secure mode to populate the store with JSON data.
2. Run the Admin CLI as the user named FTS to both register the store with the Elasticsearch cluster and create a Text Index on the data in the store.
3. Use `curl` to send secure queries to Elasticsearch to perform Full Text Search on the indexed data. See Running the Examples in Secure Mode.

Running the Examples in Secure Mode

Assuming you have deployed a secure Oracle NoSQL store and Elasticsearch cluster by executing the steps presented in Secure Elasticsearch using Sheild, Deploying and Configuring a Secure Oracle NoSQL Store, and Install the Full Text Search Public Certificate in Elasticsearch appendices, you can now execute the commands presented in this section to:

- Create and Populate a Table in the Secure Oracle NoSQL Store
- Register the Store with the Secure Elasticsearch Cluster and Create a Full Text Index
- Execute Secure Full Text Search Queries On Elasticsearch Indexed Data

Create and Populate a Table in the Secure Oracle NoSQL Store

Execute the program `LoadJsonExample` in secure mode. For example,

```
java –classpath /opt/ondb/kv/lib/kvstore.jar:src
es.table.LoadJsonExample
        –store mystore
        –host kvhost1
        –port 5000
        –file ~/examples/es/docs/senator-info.json
        -table exampleJsonTable
        -security /tmp/FTS-client.login
```

Register the Store with the Secure Elasticsearch Cluster and Create a Full Text Index

From a client node configured for secure access to the Oracle NoSQL store, start an Admin CLI for the store. For example, from the host named `clhost1`, start the CLI as the user named FTS created and provisioned as described in Deploying and Configuring a Secure Oracle NoSQL Store,

On `clhost1`

```
java -jar /opt/ondb/kv/lib/kvcli.jar runadmin
        –host kvhost1
        -port 5000
        –store mystore
        –security /tmp/FTS-client.login

  Logged in admin as FTS

kv-> plan register-es
        –clustername escluster
        –host eshost1
        –port 29100
        –secure true
        -wait

  Executed plan 25, waiting for completion...
  Plan 25 ended successfully

kv-> execute 'CREATE FULLTEXT INDEX jsonTxtIndex ON
    exampleJsonTable (
      jsonField.current{"type":"boolean"},
      jsonField.party{"type":"string","analyzer":"standard"},
      jsonField.duties.committe{"type":"string"},
      jsonField.contrib{"type":"double"})';

  Statement completed successfully

kv-> exit
```

Execute Secure Full Text Search Queries On Elasticsearch Indexed Data

From a client node configured for secure access to the Elasticsearch cluster such as the `clhost1` node presented in Secure Elasticsearch using Sheild, execute queries like the following:

On `clhost1`

```
curl –k -E /tmp/elasticsearch-eshost1.pem
        --key /tmp/elasticsearch-eshost1.pkey
        –X GET 'http://eshost1:29100/_cat/indices'

curl –k -E /tmp/elasticsearch-eshost2.pem
        --key /tmp/elasticsearch-eshost2.pkey
        –X GET 'http://eshost2:29100/
ondb.kvstore.examplejsontable.jsontxtindex/_mapping?pretty'

curl –k -E /tmp/elasticsearch-eshost3.pem
        --key /tmp/elasticsearch-eshost3.pkey
        –X GET 'http://eshost3:29100/
ondb.kvstore.examplejsontable.jsontxtindex/_search?pretty'

curl –k -E /tmp/elasticsearch-eshost1.pem
        --key /tmp/elasticsearch-eshost1.pkey
        –X GET 'http://eshost1:29100/
ondb.mystore.examplejsontable.jsontxtindex/_search?pretty'
        '-d {query":{"bool":{
                "must":{"match":{"jsonCol.party":"Democrat"}},
                "must":{"match":"jsonCol.current":"true"}},
                "must":{"range":{"jsonField.contrib":
{"gte":"1000000.00","lte":20000000.00"}}},
                "must":"match":{"jsonField.duties.committe":"Judiciary
Apropriations"}}}}}'
```

> **Note:**
>
> The queries above can be sent to any of the nodes in the Elasticsearch cluster (`eshost1`, `eshost2`, or `eshost3`). Just be sure to specify the public certificate and private key corresponding to the particular node to which you send the query.