Oracle® NoSQL Database

Integrations Guide

Release 20.1
F30919-01
June 2020

ORACLE

Oracle NoSQL Database Integrations Guide, Release 20.1

F30919-01

Contents

Part I

Integration with Elastic Search for Full Text Search

Topics

1

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.

2

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.

3
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.

4

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.

A

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.

B

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;
    }
}
```

C

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.

D

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.

E

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.

F

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.

Glossary