Oracle

Getting Started with NoSQL Database Tables API

12c Release 1

Library Version 12.1.3.0

ORACLE

NOSQL DATABASE

Legal Notice

Published 7/14/2014

Table of Contents

Preface

There are two different APIs that can be used to write Oracle NoSQL Database (Oracle NoSQL Database) applications: the original key/value API, and the tables API. This document describes how to write Oracle NoSQL Database applications using the tables API. Note that most application developers should use the tables API because it offers important features, including secondary indexes.

This document provides the concepts surrounding Oracle NoSQL Database, data schema considerations, as well as introductory programming examples.

This document is aimed at the software engineer responsible for writing an Oracle NoSQL Database application.

Conventions Used in This Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "The `KVStoreConfig()` constructor returns a `KVStoreConfig` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *KVHOME* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");
KVStore kvstore = null;
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:

```
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");
KVStore kvstore = null;

try {
    kvstore = KVStoreFactory.getStore(kconfig);
```

```
} catch (FaultException fe) {
    // Some internal error occurred. Either abort your application
    // or retry the operation.
}
```

Note

Finally, notes of special interest are represented using a note block such as this.

Chapter 1. Introduction to Oracle NoSQL Database

Welcome to Oracle NoSQL Database. Oracle NoSQL Database provides multi-terabyte distributed key/value pair storage that offers scalable throughput and performance. That is, it services network requests to store and retrieve data which is accessed as tables of information or, optionally, as key-value pairs. Oracle NoSQL Database services these types of data requests with a latency, throughput, and data consistency that is predictable based on how the store is configured.

Oracle NoSQL Database offers full Create, Read, Update and Delete (CRUD) operations with adjustable durability guarantees. Oracle NoSQL Database is designed to be highly available, with excellent throughput and latency, while requiring minimal administrative interaction.

Oracle NoSQL Database provides performance scalability. If you require better performance, you use more hardware. If your performance requirements are not very steep, you can purchase and manage fewer hardware resources.

Oracle NoSQL Database is meant for any application that requires network-accessible data with user-definable read/write performance levels. The typical application is a web application which is servicing requests across the traditional three-tier architecture: web server, application server, and back-end database. In this configuration, Oracle NoSQL Database is meant to be installed behind the application server, causing it to either take the place of the back-end database, or work alongside it. To make use of Oracle NoSQL Database, code must be written (using Java or C) that runs on the application server.

An application makes use of Oracle NoSQL Database by performing network requests against Oracle NoSQL Database's data store, which is referred to as the KVStore. The requests are made using the Oracle NoSQL Database Driver, which is linked into your application as a Java library (.jar file), and then accessed using a series of Java APIs.

By using the Oracle NoSQL Database APIs, the developer is able to perform create, read, update and delete operations on the data contained in the KVStore. The usage of these APIs is introduced later in this manual.

Note

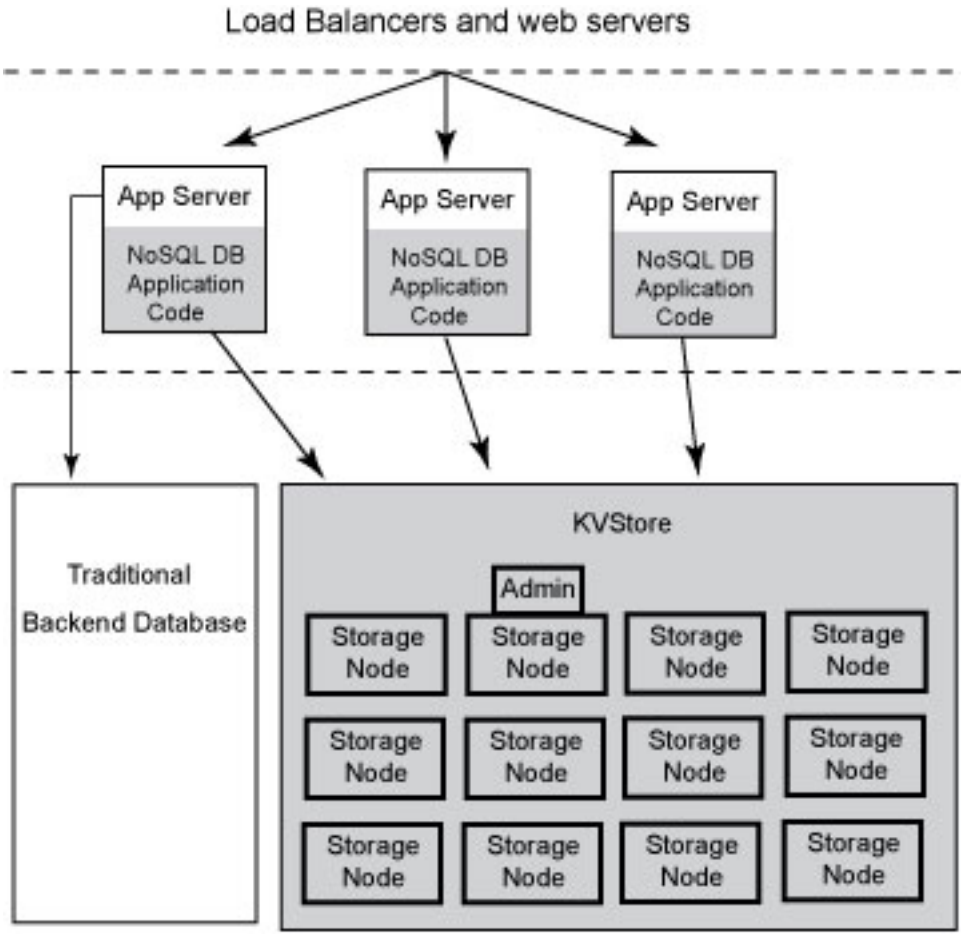
Oracle NoSQL Database is tested using Java 7, and so Oracle NoSQL Database should be used only with that version of Java.

The KVStore

The KVStore is a collection of Storage Nodes which host a set of Replication Nodes. Data is spread across the Replication Nodes. Given a traditional three-tier web architecture, the KVStore either takes the place of your back-end database, or runs alongside it.

The store contains multiple Storage Nodes. A *Storage Node* is a physical (or virtual) machine with its own local storage. The machine is intended to be commodity hardware. It should be, but is not required to be, identical to all other Storage Nodes within the store.

The following illustration depicts the typical architecture used by an application that makes use of Oracle NoSQL Database:



Every Storage Node hosts one or more Replication Nodes as determined by its *capacity*. The capacity of a Storage Node serves as a rough measure of the hardware resources associated with it. A store can consist of Storage Nodes of different capacities. Oracle NoSQL Database will ensure that a Storage Node is assigned a load that is proportional to its capacity. A Replication Node in turn contains at least one and typically many partitions. Also, each Storage Node contains monitoring software that ensures the Replication Nodes which it hosts are running and are otherwise healthy.

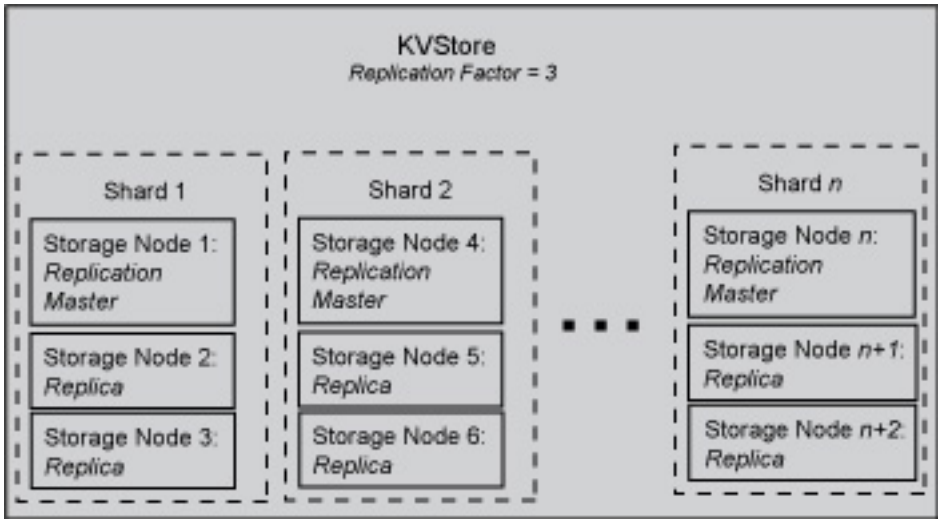
Replication Nodes and Shards

At a very high level, a *Replication Node* can be thought of as a single database which contains key-value pairs.

Replication Nodes are organized into *shards*. A shard contains a single Replication Node, called the *master* node, which is responsible for performing database writes, as well as one or more read-only *replicas*. The *master* node copies all writes to the replicas. These replicas are used to service read-only operations. Although there can be only one master node at any given

time, any of the members of the shard (with the exception of nodes in a secondary zone as described below) are capable of becoming a *master* node. In other words, each shard uses a single master/multiple replica strategy to improve read throughput and availability.

The following illustration shows how the KVStore is divided up into shards:



Note that if the machine hosting the master should fail in any way, then the master automatically fails over to one of the other nodes in the shard. That is, one of the replica nodes is automatically promoted to master.

Production KVStores should contain multiple shards. At installation time you provide information that allows Oracle NoSQL Database to automatically decide how many shards the store should contain. The more shards that your store contains, the better your write performance is because the store contains more nodes that are responsible for servicing write requests.

Replication Factor

The number of nodes belonging to a shard is called its *Replication Factor*. The larger a shard's Replication Factor, the faster its read throughput (because there are more machines to service the read requests) but the slower its write performance (because there are more machines to which writes must be copied).

Once you set the Replication Factor for each zone in the store, Oracle NoSQL Database makes sure the appropriate number of Replication Nodes are created for each shard residing in each zone making up your store. The number of copies, or replicas, maintained in a zone is called the *Zone Replication Factor*. The total number of replicas in all Primary zones is called the *Primary Replication Factor*, and the total number in all Secondary zones is called the *Secondary Replication Factor*. For all zones in the store, the total number of replicas across the entire store is called the *Store Replication Factor*.

Partitions

Each shard contains one or more *partitions*. Table rows (or key-value pairs) in the store are accessed by the data's key. Keys, in turn, are assigned to a partition. Once a key is placed in a partition, it cannot be moved to a different partition. Oracle NoSQL Database spreads records evenly across all available partitions by hashing each record's key.

As part of your planning activities, you must decide how many partitions your store should have. Note that this is not configurable after the store has been installed.

It is possible to expand and change the number of Storage Nodes in use by the store. When this happens, the store can be reconfigured to take advantage of the new resources by adding new shards. When this happens, partitions are balanced between new and old shards by redistributing partitions from one shard to another. For this reason, it is desirable to have enough partitions so as to allow fine-grained reconfiguration of the store. Note that there is a minimal performance cost for having a large number of partitions. As a rough rule of thumb, there should be at least 10 to 20 partitions per shard, and the number of partitions should be evenly divisible by the number of shards. Since the number of partitions cannot be changed after the initial deployment, you should consider the maximum future size of the store when specifying the number of partitions.

Zones

A zone is a physical location that supports good network connectivity among the Storage Nodes deployed in it and has some level of physical separation from other zones. A zone generally includes redundant or backup power supplies, redundant data communications connections, environmental controls (for example: air conditioning, fire suppression) and security devices. A zone may represent an actual physical data center building, but could also represent a floor, room, pod, or rack, depending on the particular deployment. Oracle recommends you install and configure your store across multiple zones to guard against systemic failures affecting an entire physical location, such as a large scale power or network outage.

Multiple zones provide fault isolation and increase the availability of your data in the event of a single zone failure.

Zones come in two types. *Primary* zones contain nodes which can serve as masters or replicas. Zones are created as primary zones by default. *Secondary* zones contain nodes which can only serve as replicas. Secondary zones can be used to make a copy of the data available at a distant location, or to maintain an extra copy of the data to increase redundancy or read capacity.

Topologies

A *topology* is the collection of zones, storage nodes, replication nodes and administration services that make up a NoSQL DB store. A deployed store has one topology that describes its state at a given time.

After initial deployment, the topology is laid out so as to minimize the possibility of a single point of failure for any given shard. This means that while a Storage Node might host more

than one Replication Node, those Replication Nodes will never be from the same shard. This improves the chances of the shard continuing to be available for reads and writes even in the face of a hardware failure that takes down the host machine.

Topologies can be changed to achieve different performance characteristics, or in reaction to changes in the number or characteristics of the Storage Nodes. Changing and deploying a topology is an iterative process.

Access and Security

Access to the KVStore and its data is performed in two different ways. Routine access to the data is performed using Java APIs that the application developer uses to allow his application to interact with the Oracle NoSQL Database Driver, which communicates with the store's Storage Nodes in order to perform whatever data access the application developer requires. The Java APIs that the application developer uses are introduced later in this manual.

In addition, administrative access to the store is performed using a command line interface or a browser-based graphical user interface. System administrators use these interfaces to perform the few administrative actions that are required by Oracle NoSQL Database. You can also monitor the store using these interfaces.

For most production stores, authentication over SSL is normally required by both the command line interface and the Java APIs. It is possible to install a store such that authentication is not required, but this is not recommended. For details on Oracle NoSQL Database's security features, see the *Oracle NoSQL Database Security Guide*.

Note

Oracle NoSQL Database is intended to be installed in a secure location where physical and network access to the store is restricted to trusted users. For this reason, at this time Oracle NoSQL Database's security model is designed to prevent accidental access to the data. It is *not* designed to prevent denial-of-service attacks.

KVLite

KVLite is a simplified version of Oracle NoSQL Database. It provides a single-node store that is not replicated. It runs in a single process without requiring any administrative interface. You configure, start, and stop KVLite using a command line interface.

KVLite is intended for use by application developers who need to unit test their Oracle NoSQL Database application. It is not intended for production deployment, or for performance measurements.

KVLite is installed when you install KVStore. It is available in the `kvstore.jar` file in the `lib` directory of your Oracle NoSQL Database distribution.

Note that KVLite cannot be configured as a secure store.

For more information on KVLite, see Introduction to Oracle KVLite (page 7).

The Administration Command Line Interface history

By default Oracle NoSQL Database uses the Java Jline library to support saveable command line history in the CLI. If you want to disable this feature, the following Java property should be set while running runadmin:

```
java -Doracle.kv.shell.jline.disable=true -jar KVHOME/kvstore.jar \
runadmin -host <hostname> -port <portname>
```

Command line history is saved to a file so that it is available after restart.
By default, Oracle NoSQL Database attempts to save the history in a KVHOME/.jlineoracle.kv.impl.admin.client.CommandShell.history file, which is created and opened automatically. The default history saved is 500 lines.

> **Note**
> If the history file cannot be opened, it will fail silently and the CLI will run without saved history.

The default history file path can be overridden by setting the `oracle.kv.shell.history.file="path"` Java property.

The default number of lines to save to the file can be modified by setting the `oracle.kv.shell.history.size=<int_value>` Java property.

The Administration Command Line Interface

The Administration command line interface (CLI) is the primary tool used to manage your store. It is used to configure, deploy, and change store components.

The CLI can also be used to get, put, and delete store records or tables, retrieve schema, and display general information about the store. It can also be used to diagnose problems or potential problems in the system, fix actual problems by adding, removing or modifying store data and/or verify that the store has data. It is particularly well-suited for a developer who is creating a Oracle NoSQL Database application, and who needs to either populate a store a with a small number of records so as to have data to develop against, or to examine the store's state as part of development debugging activities.

The command line interface is accessed using the following command:

```
java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar runadmin
```

For a listing of all the commands available to you in the CLI, see appendix A of the *Oracle NoSQL Database Administrator's Guide*.

Chapter 2. Introduction to Oracle KVLite

KVLite is a single-node, single shard store. It usually runs in a single process and is used to develop and test client applications. KVLite is installed when you install Oracle NoSQL Database.

Note

KVLite supports only non-authenticated access to the store. That is, you cannot configure KVLite such that your code can authenticate, or log in, to it. If you are developing code for a store that requires authentication, then you must install a test store that is configured for authentication access in the same way as your production store.

Using the authentication APIs is described in Using the Authentication APIs (page 13). For information on configuring a store to require authentication, see the *Oracle NoSQL Database Security Guide*.

Starting KVLite

You start KVLite by using the `kvlite` utility, which can be found in `KVHOME/lib/kvstore.jar`. If you use this utility without any command line options, then KVLite will run with the following default values:

- The store name is `kvstore`.
- The hostname is the local machine.
- The registry port is `5000`.
- The directory where Oracle NoSQL Database data is placed (known as KVROOT) is `./kvroot`.
- The administration process is turned on using port 5001.

This means that any processes that you want to communicate with KVLite can only connect to it on the local host (127.0.0.1) using port 5000. If you want to communicate with KVLite from some machine other than the local machine, then you must start it using non-default values. The command line options are described later in this chapter.

For example:

```
> java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar kvlite
```

Note

To avoid using too much heap space, you should specify -Xmx and -Xms flags for Java when running administrative and utility commands.

When KVLite has started successfully, it issues one of two statements to stdout, depending on whether it created a new store or is opening an existing store:

```
Created new kvlite store with args:
-root ./kvroot -store <kvstore name> -host <localhost> -port 5000
-admin 5001
```

or

```
Opened existing kvlite store with config:
-root ./kvroot -store <kvstore name> -host <localhost> -port 5000
-admin 5001
```

where <kvstore name> is the name of the store and <localhost> is the name of the local host. It takes about 10 - 60 seconds before this message is issued, depending on the speed of your machine.

Note that you will not get the command line prompt back until you stop KVLite.

Stopping and Restarting KVLite

To stop KVLite, use ^C from within the shell where KVLite is running.

To restart the process, simply run the `kvlite` utility without any command line options. Do this even if you provided non-standard options when you first started KVLite. This is because KVLite remembers information such as the port value and the store name in between run times. You cannot change these values by using the command line options.

If you want to start over with different options than you initially specified, delete the KVROOT directory (`./kvroot`, by default), and then re-run the `kvlite` utility with whatever options you desire. Alternatively, specify the `-root` command line option, making sure to specify a location other than your original KVROOT directory, as well as any other command line options that you want to change.

Verifying the Installation

There are several things you can do to verify your installation, and ensure that KVLite is running:

- Start another shell and run:

  ```
  jps -m
  ```

  The output should show KVLite (and possibly other things as well, depending on what you have running on your machine).

- Run the `kvclient` test application:

  1. cd KVHOME
  2. java -Xmx256m -Xms256m -jar lib/kvclient.jar

  This should write the release to stdout:

  ```
  12cR1.M.N.O...
  ```

- Compile and run the example program:

  1. cd KVHOME
  2. Compile the example:

     ```
     javac -g -cp lib/kvclient.jar:examples examples/hello/*.java
     ```

  3. Run the example using all default parameters:

     ```
     java -Xmx256m -Xms256m \
     -cp lib/kvclient.jar:examples hello.HelloBigDataWorld
     ```

     Or run it using non-default parameters, if you started KVLite using non-default values:

     ```
     java -Xmx256m -Xms256m \
     -cp lib/kvclient.jar:examples hello.HelloBigDataWorld \
         -host <hostname> -port <hostport> -store <kvstore name>
     ```

kvlite Utility Command Line Parameter Options

This section describes the command line options that you can use with the `kvlite` utility.

Note that you can only specify these options the first time KV Lite is started. Most of the parameter values specified here are recorded in the KVHOME directory, and will be used when you restart the KVLite process regardless of what you provide as command line options. If you want to change your initial values, either delete your KVHOME directory before starting KV Lite again, or specify the `-root` option (with a different KVHOME location than you initially used) when you provide the new values.

- `-admin <port>`

  If this option is specified, the administration user interface is started. The port identified here is the port you use to connect to the UI.

- `-help`

  Print a brief usage message, and exit.

- `-host <hostname>`

  Identifies the name of the host on which KVLite is running. Use this option ONLY if you are creating a new store.

  If you want to access this instance of KVLite from remote machines, supply the local host's real hostname. Otherwise, specify `localhost` for this option.

- `-logging`

  Turns on Java application logging. The log files are placed in the examples directory in your Oracle NoSQL Database distribution.

- `-port <port>`

Identifies the port on which KVLite is listening for client connections. Use this option ONLY if you are creating a new store.

- `-root <path>`

  Identifies the path to the Oracle NoSQL Database home directory. This is the location where the store's database files are contained. The directory identified here must exist. If the appropriate database files do not exist at the location identified by the option, they are created for you.

- `-store <storename>`

  Identifies the name of a new store. Use this option ONLY if you are creating a new store.

Chapter 3. Developing for Oracle NoSQL Database

You access the data in the Oracle NoSQL Database KVStore using Java APIs that are provided with the product. There are two different programming paradigms that you can use to write your Oracle NoSQL Database applications. One uses a tables API, and the other offers a key/value pair API. Most customers should use the tables API because it offers important features that do not appear in the key/value pair API, such as secondary indexing. However, as of this release, the tables API does not allow you to insert Large Objects into a table. You can, however, use the LOB API alongside of the tables API.

Regardless of the API you decide to use, the provided classes and methods allow you to write data to the store, retrieve it, and delete it. You use these APIs to define consistency and durability guarantees. It is also possible to execute a sequence of store operations atomically so that all the operations succeed, or none of them do.

The rest of this book introduces the Java APIs that you use to access the store, and the concepts that go along with them. This book describes the tables API. If you prefer to use the key/value API, you should read *Oracle NoSQL Database Getting Started with the Key/Value API*.

Note

Oracle NoSQL Database is tested with Java 7, and so Oracle NoSQL Database should be used only with that version of Java.

The KVStore Handle

In order to perform store access of any kind, you must obtain a `KVStore` handle. You obtain a KVStore handle by using the `KVStoreFactory.getStore()` method.

When you get a `KVStore` handle, you must provide a `KVStoreConfig` object. This object identifies important properties about the store that you are accessing. We describe the `KVStoreConfig` class next in this chapter, but at a minimum you must use this class to identify:

- The name of the store. The name provided here must be identical to the name used when the store was installed.
- The network contact information for one or more helper hosts. These are the network name and port information for nodes currently belonging to the store. Multiple nodes can be identified using an array of strings. You can use one or many. Many does not hurt. The downside of using one is that the chosen host may be temporarily down, so it is a good idea to use more than one.

In addition to the `KVStoreConfig` class object, you can also provide a `PasswordCredentials` class object to `KVStoreFactory.getStore()`. You do this if you are using a store that has been configured to require authentication. See Using the Authentication APIs (page 13) for more information.

For a store that does not require authentication, you obtain a store handle like this:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...

String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);
KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

The KVStoreConfig Class

The `KVStoreConfig` class is used to describe properties about a `KVStore` handle. Most of the properties are optional; those that are required are provided when you construct a class instance.

The properties that you can provide using `KVStoreConfig` are:

- Consistency

  Consistency is a property that describes how likely it is that a record read from a replica node is identical to the same record stored on a master node. For more information, see Consistency Guarantees (page 74).

- Durability

  Durability is a property that describes how likely it is that a write operation performed on the master node will not be lost if the master node is lost or is shut down abnormally. For more information, see Durability Guarantees (page 81).

- Helper Hosts

  Helper hosts are hostname/port pairs that identify where nodes within the store can be contacted. Multiple hosts can be identified using an array of strings. Typically an application developer will obtain these hostname/port pairs from the store's deployer and/or administrator. For example:

  ```
  String[] hhosts = {"n1.example.org:3333", "n2.example.org:3333"};
  ```

- Request Timeout

  Configures the amount of time the `KVStore` handle will wait for an operation to complete before it times out.

- Store name

  Identifies the name of the store.

- Password credentials and optionally a reauthentication handler

See the next section on authentication.

Using the Authentication APIs

Oracle NoSQL Database can be installed such that your client code does not have to authenticate to the store. (For the sake of clarity, most of the examples in this book do not perform authentication.) However, if you want your store to operate in a secure manner, you can require authentication. Note that doing so will result in a performance cost due to the overhead of using SSL and authentication. While best practice is for a production store to require authentication over SSL, some sites that are performance sensitive may want to forgo that level of security.

Authentication involves sending username/password credentials to the store at the time a store handle is acquired. A store that is configured to support authentication is automatically configured to communicate with clients using SSL in order to ensure privacy of the authentication and other sensitive information. When SSL is used, SSL certificates need to be installed on the machines where your client code runs in order to validate that the store that is being accessed is trustworthy.

Configuring a store for authentication is described in the *Oracle NoSQL Database Security Guide*.

Configuring SSL

If you are using a secure store, then all communications between your client code and the store is transported over SSL, including your authentication credentials. You must therefore configure your client code to use SSL. To do this, you identify where the SSL certificate data is, and you also separately indicate that the SSL transport is to be used.

Identifying the Trust Store

When an Oracle NoSQL Database store is configured to use the SSL transport, a series of security files are generated using a security configuration tool. One of these files is the `client.trust` file, which must be copied to any machine running Oracle NoSQL Database client code.

For information on using the security configuration tool, see the *Oracle NoSQL Database Security Guide*.

Your code must be told where the `client.trust` file can be found because it contains the certificates necessary for your client to establish an SSL connection with the store. You indicate where this file is physically located on your machine using the `oracle.kv.ssl.trustStore` property. There are two ways to set this property:

1. Identify the location of the trust store by using a `Properties` object to set the `oracle.kv.ssl.trustStore` property. You then use `KVStoreConfig.setSecurityProperties()` to pass the `Properties` object to your `KVStore` handle.

When you use this method, you use `KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY` as the property name.

2. Use the `oracle.kv.security` property to refer to a properties file, such as the `client.trust` file. In that file, set the `oracle.kv.ssl.trustStore` property.

Setting the SSL Transport Property

In addition to identifying the location of the `client.trust` file, you must also tell your client code to use the SSL transport. You do this by setting the `oracle.kv.transport` property in one of two ways:

1. Identify the location of the trust store by using a `Properties` object to set the `oracle.kv.transport` property. You then use `KVStoreConfig.setSecurityProperties()` to pass the `Properties` object to your `KVStore` handle.

   When you use this method, you use `KVSecurityConstants.TRANSPORT_PROPERTY` as the property name, and `KVSecurityConstants.SSL_TRANSPORT_NAME` as the property value.

2. Use the `oracle.kv.security` property to refer to a properties file, such as the `client.trust` file. In that file, set the `oracle.kv.transport` property.

Authenticating to the Store

You authenticate to the store by specifying a `LoginCredentials` implementation instance to `KVStoreFactory.getStore()`. Oracle NoSQL Database provides the `PasswordCredentials` class as a `LoginCredentials` implementation. If your store requires SSL to be used as the transport, configure that prior to performing the authentication. (See the previous section for details.)

Your code should be prepared to handle a failed authentication attempt. `KVStoreFactory.getStore()` will throw `AuthenticationFailure` in the event of a failed authentication attempt. You can catch that exception and handle the problem there.

The following is a simple example of obtaining a store handle for a secured store. The SSL transport is used in this example.

```
import java.util.Properties;

import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;



KVStore store = null;
```

```
try {
    /*
     * storeName, hostName, port, username, and password are all
     * strings that would come from somewhere else in your
     * application.
     */
    KVStoreConfig kconfig =
        new KVStoreConfig(storeName, hostName + ":" + port);

    /* Set the required security properties */
    Properties secProps = new Properties();
    secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                         KVSecurityConstants.SSL_TRANSPORT_NAME);
    secProps.setProperty
        (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
        "/home/kv/client.trust");
    kconfig.setSecurityProperties(secProps);

    store =
        KVStoreFactory.getStore(kconfig,
            new PasswordCredentials(username,
                                    password.toCharArray()));
            null /* ReauthenticateHandler */);
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Another way to handle the login is to place your authentication credentials in a flat text file that contains all the necessary properties for authentication. In order for this to work, a password store must have been configured for your Oracle NoSQL Database store. (See the *Oracle NoSQL Database Security Guide* for information on setting up password stores).

For example, suppose your store has been configured to use a password file password store and it is contained in a file called `login.pwd`. In that case, you might create a login properties file called `login.txt` that looks like this:

```
oracle.kv.auth.username=clientUID1
oracle.kv.auth.pwdfile.file=/home/nosql/login.pwd
oracle.kv.transport=ssl
oracle.kv.ssl.trustStore=/home/nosql/client.trust
```

In this case, you can perform authentication in the following way:

```
import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
```

```
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

/* the client gets login credentials from the login.txt file */
/* can be set on command line as well */
System.setProperty("oracle.kv.security", "/home/nosql/login.txt");

KVStore store = null;
try {
    /*
     * storeName, hostName, port are all strings that would come
     * from somewhere else in your application.
     *
     * Notice that we do not pass in any login credentials.
     * All of that information comes from login.txt
     */
    myStoreHandle =
        KVStoreFactory.getStore(
            new KVStoreConfig(storeName, hostName + ":" + port))
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Renewing Expired Login Credentials

It is possible for an authentication session to expire. This can happen for several reasons. One is that the store's administrator has configured the store to not allow session extension and the session has timed out. These properties are configured using `sessionExtendAllow` and `sessionTimeout`. See the *Oracle NoSQL Database Security Guide* for information on these properties.

Reauthentication might also be required if some kind of a major disruption has occurred to the store which caused the authentication session to become invalidated. This is a pathological condition which you should not see with any kind of frequency in a production store. Stores which are installed in labs might exhibit this condition more, especially if the stores are frequently restarted.

An application can encounter an expired authentication session at any point in its lifetime, so robust code that must remain running should always be written to respond to authentication session expirations.

When an authentication session expires, by default the method which is attempting store access will throw `AuthenticationRequiredException`. Upon seeing this, your code needs to reauthenticate to the store, and then retry the failed operation.

You can manually reauthenticate to the store by using the `KVStore.login()` method. This method requires you to provide the login credentials via a `LoginCredentials` class instance (such as `PasswordCredentials`):

```
try {
    ...
    /* Store access code happens here */
    ...
} catch (AuthenticationRequiredException are) {
    /*
     * myStoreHandle is a KVStore class instance.
     *
     * pwCreds is a PasswordCredentials class instance, obtained
     * from somewhere else in your code.
     */
    myStoreHandle.login(pwCreds);
}
```

Note that this is not required if you use the `oracle.kv.auth.username` and `oracle.kv.auth.pwdfile.file` properties, as shown in the previous section. In that case, your Oracle NoSQL Database client code will automatically and silently reauthenticate your client using the values specified by those properties.

A third option is to create a `ReauthenticationHandler` class implementation that performs your reauthentication for you. This option is only necessary if you provided a `LoginCredentials` implementation instance (that is, `PasswordCredentials`) in a call to `KVStoreFactory.getStore()`, and you want to avoid a subsequent need to retry operations by catching `AuthenticationRequiredException`.

A truly robust example of a `ReauthenticationHandler` implementation is beyond the scope of this manual (it would be driven by highly unique requirements that are unlikely to be appropriate for your site). Still, in the interest of completeness, the following shows a very simple and not very elegant implementation of `ReauthenticationHandler`:

```
package kvstore.basicExample

import oracle.kv.ReauthenticationHandler;
import oracle.kv.PasswordCredentials;

public class MyReauthHandler implements ReauthenticationHandler {
    public void reauthenticate(KVStore reauthStore) {
        /*
         * The code to obtain the username and password strings would
         * go here. This should be consistent with the code to perform
         * simple authentication for your client.
         */
        PasswordCredentials cred = new PasswordCredentials(username,
            password.toCharArray());

        reauthStore.login(cred);
```

```
    }
}
```

You would then supply a `MyReauthHandler` instance when you obtain your store handle:

```
import java.util.Properties;

import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import kvstore.basicExample.MyReauthHandler;

KVStore store = null;
try {
    /*
     * storeName, hostName, port, username, and password are all
     * strings that would come from somewhere else in your
     * application. The code you use to obtain your username
     * and password should be consistent with the code used to
     * obtain that information in MyReauthHandler.
     */
    KVStoreConfig kconfig =
        new KVStoreConfig(storeName, hostName + ":" + port);

    /* Set the required security properties */
    Properties secProps = new Properties();
    secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                         KVSecurityConstants.SSL_TRANSPORT_NAME);
    secProps.setProperty
        (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
        "/home/kv/client.trust");
    kconfig.setSecurityProperties(secProps);

    store =
        KVStoreFactory.getStore(kconfig,
            new PasswordCredentials(username,
                                    password.toCharArray()));
            new MyReauthHandler());
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Unauthorized Access

Currently, clients which must authenticate to a store either have no access or complete access, depending on the current state of their authorization session. However, future releases of Oracle NoSQL Database may introduce authorization levels such that a call to the Oracle NoSQL Database API could fail due to not having the authority to perform the operation. When this happens, `UnauthorizedException` will be thrown.

In this release, your code should never see `UnauthorizedException`. However, you might want to plan your code so that it is possible to respond gracefully to this exception, especially if you think your organization would be interested in using multi-level access controls.

When `UnauthorizedException` is seen, the operation should not be retried. Instead, the operation should either be abandoned entirely, or your code could attempt to reauthenticate using different credentials that would have the required permissions necessary to perform the operation. Note that a client can log out of a store using `KVStore.logout()`. How your code logs back in is determined by how your store is configured for access, as described in the previous sections.

Chapter 4. Introducing Oracle NoSQL Database Tables and Indexes

The Oracle NoSQL Database tables API is the recommended method of coding an Oracle NoSQL Database client application. It allows you to manipulate data using a tables metaphor, in which data is organized in multiple columns of data. An unlimited number of subtables are supported by this API. You can also create indexes to improve query speeds against your tables.

Note

You should avoid any possibility of colliding keys if your store is accessed by a mix of clients that use both the tables and the key/value APIs.

Table creation, deletion and evolution is performed using the command line interface (CLI). You use the CLI to define tables, including the data types supported by each column in the table. You also use the CLI to define indexes. Once you have created your table(s) using the CLI, you then use the tables API to read and write the data stored in those tables.

Defining Tables

Before an Oracle NoSQL Database client can read or write to a table in the store, the table must be created using the command line interface (CLI). While it is possible to define tables interactively using the CLI, doing so represents challenges when developing table definitions for use in an enterprise environment. In order to avoid typos as code moves from the development environment, to test, to production, it is best to perform table definitions using a script.

A CLI script is simply a series of CLI commands contained in a single text file, one command to a line. To run the script, you start the CLI and then use the `load` command.

For example, suppose you wanted to use a table named `myTable` with four columns per row: `item`, `count1`, `count2`, and `percentage`. To create this table, you can log into the CLI and use the `table create`, `add-field`, `primary-key`, and `plan add-table` commands interactively. Or you can just collect all these commands into a plain text file like this:

```
## Enter into table creation mode
## This enters into a submode for the CLI, which offers
## commands specifically used to define tables.
table create -name myTable
## Now add the fields
add-field -type STRING -name item
add-field -type STRING -name description
add-field -type INTEGER -name count
add-field -type DOUBLE -name percentage
## A primary key must be defined for every table
## Here, we will define field 'item' as the primary key.
primary-key -field item
## Exit table creation mode, returning to the
```

```
## main CLI commands.
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myTable -wait
```

Note

Primary keys are a concept that have not yet been introduced in this manual. See Primary and Shard Key Design (page 33) for a complete explanation on what they are and how you should use them.

To run the script, start the CLI and then use the `load` command. Suppose you placed the above script into a file named `createTable.txt`:

```
> java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar runadmin -host <hostName> \
-port <port> -store <storeName>
kv-> load -file createTable.txt
Table myTable built
Executed plan 8, waiting for completion...
Plan 8 ended successfully

kv->
```

Note

The above example assumes you are connecting to a nonsecure store. If you are using a secure store, then you will have to authenticate when you start the CLI. You do this using the `-admin-security` command line option.

By performing all your table and index manipulation in this way, you can ensure a consistent store environment all the way through your development/test/product deployment cycle.

Name Limitations

Table, index and field names are case-preserving, but case-insensitive. So you can, for example, create a field named `MY_NAME`, and later reference it as `my_name` without error. However, whenever the field name is displayed, it will display as `MY_NAME`.

Table and index names are limited to 32 characters. Field names can be 64 characters. All table, index and field names are restricted to alphanumeric characters, plus underscore ("_"). All names must start with a letter.

Supported Table Data Types

You specify schema for each column in an Oracle NoSQL Database table. This schema can be a primitive data type, or complex data types that are handled as objects.

Supported simple data types are:

- Binary

  Implemented as a Java byte array with no predetermined fixed size.

- Boolean
- Double
- Float
- Integer
- Long
- Java String

Complex data types are non-atomic in nature:

- Array

  An array of values, all of the same type.

- Enum

  An enumeration, represented as an array of strings.

- Fixed Binary

  A fixed-sized binary type (Java byte array) used to handle binary data where each record is the same size. It uses less storage than an unrestricted binary field, which requires the length to be stored with the data.

- Map

  An unordered map type where all entries are constrained by a single type.

- Records

  See the following section.

Record Fields

As described in Defining Child Tables (page 27), you can create child tables to hold subordinate information, such as addresses in a contacts database, or vendor contact information for an inventory system. When you do this, you can create an unlimited number of rows in the child table, and you can index the fields in the child table's rows.

However, child tables are not required in order to organize subordinate data. If you have very simple requirements for subordinate data, and you do not want to index the fields in the subordinate data, you can use record fields instead of a child tables. In general, you can use record fields instead of child tables if you only want a fixed, small number of instances of the record for each parent table row. For anything beyond trivial cases, you should use child tables. (Note that there is no downside to using child tables even for trivial cases.)

The assumption when using record fields is that you have a fixed known number of records that you will want to manage (unless you organize them as arrays). For example, for a contacts database, child tables allows you to have an unlimited number of addresses associated for each user. But by using records, you can associate a fixed number of addresses by creating a record field for each supported address (home and work, for example).

You create a record field using `add-record-field`, which puts you into a special submode within the CLI that you must exit or quit just as you do when creating a table:

```
## Enter into table creation mode
table create -name myContactsTable
## Now add the fields
add-field -type STRING -name uid
add-field -type STRING -name surname
add-field -type STRING -name familiarName
add-field -type STRING -name homePhone
add-field -type STRING -name workPhone

## Create a record field. This puts us into a new submode.
add-record-field -name homeAddress
add-field -type STRING -name street
add-field -type STRING -name city
add-field -type STRING -name state
add-field -type INTEGER -name zip -min 00000 -max 99999
### Exit record field creation mode
exit

## Add a second record field
add-record-field -name workAddress
add-field -type STRING -name street
add-field -type STRING -name city
add-field -type STRING -name state
add-field -type INTEGER -name zip -min 00000 -max 99999
### Exit record field creation mode
exit

## A primary key must be defined for every table
primary-key -field uid
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myContactsTable -wait
```

Alternatively, you can create an array of record fields. This allows you to create an unlimited number of address records per field. Note, however, that in general you should use child tables in this case.

```
## Enter into table creation mode
table create -name myContactsTable
```

```
## Now add the fields
add-field -type STRING -name uid
add-field -type STRING -name surname
add-field -type STRING -name familiarName
add-field -type STRING -name homePhone
add-field -type STRING -name workPhone

## Create an array field. This puts us into a new submode.
add-array-field -name addresses

## Create a record field for the array.
## This puts us into a new submode.
add-record-field -name address
add-field -type ENUM -name addressType -enum-values home,work,other
add-field -type STRING -name street
add-field -type STRING -name city
add-field -type STRING -name state
add-field -type INTEGER -name zip -min 00000 -max 99999
### Exit record field creation mode
exit

### Exit array field creation mode
exit

## A primary key must be defined for every table
primary-key -field uid
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myContactsTable -wait
```

Defining Tables using Existing Avro Schema

If you are a user of the key/value API, then you probably have been using Avro schema to describe your record values. You can create a table based on Avro schema which currently exists in your store, and in so doing overlay the existing store records. You can then operate on that data using both the tables API and the key/value API so long as you do not evolve (change) the table definitions. This is intended as a migration aid from the key/value API to the tables API.

For example, suppose you have the following Avro schema defined in your store:

```
kv-> show schema -name com.example.myItemRecord
{
  "type" : "record",
  "name" : "myItemRecord",
  "namespace" : "com.example",
  "fields" : [ {
```

```
    "name" : "itemType",
    "type" : "string",
    "default" : ""
  }, {
    "name" : "itemCategory",
    "type" : "string",
    "default" : ""
  }, {
    "name" : "itemClass",
    "type" : "string",
    "default" : ""
  }, {
    "name" : "itemColor",
    "type" : "string",
    "default" : ""
  }, {
    "name" : "itemSize",
    "type" : "string",
    "default" : ""
  }, {
    "name" : "price",
    "type" : "float",
    "default" : 0.0
  }, {
    "name" : "inventoryCount",
    "type" : "int",
    "default" : 0
  } ]
}
```

Then you can define a table using this schema. Note that the table's name must correspond directly to the first component of the key/value applications's keys.

```
kv-> table create -name myItemTable
myItemTable-> add-schema -name com.example.myItemRecord
myItemTable-> show
{
  "type" : "table",
  "name" : "myItemTable",
  "id" : "myItemTable",
  "r2compat" : true,
  "description" : null,
  "shardKey" : [ ],
  "primaryKey" : [ ],
  "fields" : [ {
    "name" : "itemType",
    "type" : "STRING"
  }, {
    "name" : "itemCategory",
    "type" : "STRING"
```

```
  }, {
    "name" : "itemClass",
    "type" : "STRING"
  }, {
    "name" : "itemColor",
    "type" : "STRING"
  }, {
    "name" : "itemSize",
    "type" : "STRING"
  }, {
    "name" : "price",
    "type" : "FLOAT",
    "default" : 0.0
  }, {
    "name" : "inventoryCount",
    "type" : "INTEGER"
  } ]
}
myItemTable->
```

At this point, you need to define your primary keys and, optionally, your shard keys in the same way you would any table. You also need to add the table to the store in the same way as always.

Note that in this case, the primary keys must be of type STRING and must also correspond to the key components used by the key/value application.

```
myItemTable->primary-key -field itemType -field itemCategory
myItemTable->exit
kv->plan add-table -name myItemTable -wait
```

Tables Compatible with Key-Only Entries (-r2-compat)

If you are a user of the key/value API, you might have created store entries that have only keys. These entries have no schema. In fact, they have no data of any kind. In this case, you can create tables that are compatible with these legacy entries using the `table create` command's `-r2-compat` flag.

For example, suppose you have key-only entries of the format:

```
/User/<id>
```

where `<id>` is a unique string ID. You can create a table to overlay this key space by doing this:

```
kv-> table create -name User -r2-compat
User-> add-field -name id -type String
User-> primary-key -field id
User-> exit
Table User built.
kv-> plan add-table -name User -wait
```

If you did not use the `-r2-compat` flag, the underlying keys generated for the table's entries would start with something other than `User`.

Note that when you create tables using existing Avro schema, the `-r2-compat` flag is automatically used.

Also note that as is the case when generating tables using Avro schema, the overlay only works so long as you do not evolve the tables.

Defining Child Tables

Oracle NoSQL Database tables can be organized in a parent/child hierarchy. There is no limit to how many child tables you can create, nor is there a limit to how deep the child table nesting can go.

Child tables are not retrieved when you retrieve a parent table, nor is the parent retrieved when you retrieve a child table.

To create a child table, you name the table using the format: <*parentTableName*>.<*childTableName*>. For example, we previously showed how to create a trivial table called `myTable`:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name itemCategory
add-field -type STRING -name description
## A primary key must be defined for every table
## Here, we will define field 'itemCategory' as the primary key.
primary-key -field itemCategory
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myTable -wait
```

We can create a child table called `myChildTable` in the following way:

```
## Enter into table creation mode
## This 'table create' will fail if 'plan add-table -name myTable'
## is not run first.
table create -name myTable.myChildTable
## Now add the fields
add-field -type STRING -name itemSKU
add-field -type STRING -name itemDescription
add-field -type FLOAT -name price
add-field -type INTEGER -name inventoryCount
## Define the primary key
primary-key -field itemSKU
## Exit table creation mode
exit
## Add the table to the store.
plan add-table -name myTable.myChildTable -wait
```

Note that when you do this, the child table inherits the parent table's primary key. In this trivial case, the child table's primary key is actually the two fields: `itemCategory` and `itemSKU`. This has several ramifications, one of which is that the parent's primary key fields are retrieved when you retrieve the child table. See Retrieve a Child Table (page 46) for more information.

Table Evolution

In the event that you must update your application at some point after it goes into production, there is a good chance that your tables will also have to be updated to either use new fields or remove existing fields that are no longer in use. You do this through the use of the `table evolve` and `plan evolve-table` commands.

Note that you cannot remove a field if it is a primary key field. You also cannot add primary key field during table evolution.

Tables can only be evolved if they have already been added to the store using either the `plan add-table` or `plan evolve-table` commands.

For example, the following script evolves the table that was created in the previous section. It adds a field and deletes another one. Again, we use a script to evolve our table so as to ensure consistency across engineering, test and production.

```
## Enter into table evolution mode
table evolve -name myTable
## Add a field
add-field -type STRING -name itemCategory
## Remove a field.
remove-field -name percentage
## Exit table creation mode
exit
plan evolve-table -name myTable -wait
```

Table Manipulation Commands

This section briefly describes the CLI commands that you use to manipulate tables. This section is for advertisement purposes only.

An exhaustive list of all CLI commands, and their syntax, can be found in the KVStore Command Reference. You can also see the command syntax for all CLI commands using the CLI `help` command.

- `plan add-table`

  Adds a table to the store that has been created but not yet added. Use the `table create` command to create the table.

- `plan evolve-table`

  Adds a table to the store that has been evolved using the `table evolve` command.

- plan remove-table

Removes an existing table from the store.

- show tables

  Shows all tables and child tables that have been added to the store.

- table clear

  Clears a table of all schema. This command works only on tables that have been created using the `table create` command, but not yet added to the store using the `plan add-table` command.

- table create

  Enters into table creation mode in which you can design the schema for the table. This mode offers a series of sub-commands:

  - add-array-field

    Adds a field to the table that accepts an array.

  - add-field

    Adds a field that accepts data of a simple type. You must identify the data's type. For example, INTEGER, LONG, DOUBLE, etc.

  - add-map-field

    Adds a field that accepts a map.

  - add-schema

    Build the table using specified Avro schema.

  - cancel

    Cancels the table creation. This returns you to the main admin prompt, abandoning any work that you might have performed when creating the table.

  - exit

    Saves the work you performed when creating the table, and returns you to the main admin prompt. After exiting table creation mode, you must issue the `plan add-table` command to add the table to your store.

  - primary-key

    The identified field is the table's primary key. Every table must have at least one field identified as its primary key. See Primary and Shard Key Design (page 33) for more information.

  - remove-field

Removes the identified field from the table.

- set-description

  Sets a plain-text description of the table. Use this to document what the table is used for.

- shard-key

  Sets a shard key for the table. See Primary and Shard Key Design (page 33) for more information.

- show

  Shows the current fields defined for the table.

- table evolve

  Allows you to evolve (modify) a table that has been added to the store. This command puts you into a special mode that allows you to add and remove fields in much the same way as you can when using `table create`. Upon completing this command, you must add your evolved table to the store using the `plan evolve-table` command.

- table list

  Lists all the table that have been built but not yet created or evolved using `plan add-table` or `plan evolve-table`..

Creating Indexes

Indexes represent an alternative way of retrieving table rows. Normally you retrieve table rows using the row's primary key. By creating an index, you can retrieve rows with dissimilar primary key values, but which share some other characteristic.

Indexes can be created on any field which is an indexable datatype, including primary key fields. See Indexable Field Types (page 31) for information on the types of fields that can be indexed.

For example, if you had a table representing types of automobiles, the primary keys for each row might be the automobile's manufacturer and model type. However, if you wanted to be able to query for all automobiles that are painted red, regardless of the manufacturer or model type, you could create an index on the table's field that contains color information.

As with tables, the best way to create indexes is to use a script. In this way, you can ensure consistency in the index(es) created across all phases of your product development cycle — from engineering, to test, to production.

Indexes can take a long time to create because Oracle NoSQL Database must examine all of the data contained in the relevant table in your store. The smaller the data contained in the table, the faster your index creation will complete. Conversely, if a table contains a lot of data, then it can take a long time to create indexes for it.

To create an index, use the `plan add-index` command. Similarly, to remove an index use the `plan remove-index` command.

To add an index, specify the table you want to index, and one or more fields to index. For example, in Defining Tables (page 20) we created a table named `myTable`. We could add a command to the end of that script that creates an index for the `percentage` field, like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name item
add-field -type STRING -name description
add-field -type INTEGER -name count
add-field -type DOUBLE -name percentage
## A primary key must be defined for every table
## Here, we will define field 'item' as the primary key.
primary-key -field item
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myTable -wait

plan add-index -name percent_idx -table myTable -field percentage -wait
```

Indexable Field Types

Fields can be indexed only if they are declared to be one of the following types:

- Integer
- Long
- Float
- Double
- String
- Enum
- Array

  In the case of arrays, the field can be indexed only if the array contains values that are of one of the other indexable types. For example, you can create an index on an array of Integers. You can also create an index on a specific record in an array of records. Only one array can participate in an index, otherwise the size of the index can grow exponentially because there is an index entry for each array entry.

Index Manipulation Commands

There are two CLI commands that you use to manipulate indexes. They are:

- `plan add-index`

  Adds an index to the store. Requires a table name and one or more field names to index. This plan can take a long time to complete if there is a large amount of existing data to be indexed.

- plan remove-index

  Removes an existing index from the store.

An exhaustive list of all CLI commands, and their syntax, can be found in KVStore Command Reference. You can also see the command syntax for all CLI commands using the CLI `help` command.

Chapter 5. Primary and Shard Key Design

Primary keys and *shard keys* are important concepts for your table design. What you use for primary and shard keys has implications in terms of your ability to read multiple rows at a time. But beyond that, your key design has important performance implications.

Primary Keys

Every table must have one or more fields designated as the primary key. This designation occurs at the time that the table is created, and cannot be changed after the fact. A table's primary key uniquely identifies every row in the table. In the simplest case, it is used to retrieve a specific row so that it can be examined and/or modified.

For example, a table might have five fields: `productName`, `productType`, `color`, `size`, and `inventoryCount`. To retrieve individual rows from the table, it might be enough to just know the product's name. In this case, you would set the primary key field as `productName` and then retrieve rows based on the product name that you want to examine/manipulate.

In this case, the CLI script that you would use to create this table might be:

```
## Enter into table creation mode
table create -name myProducts
## Now add the fields
add-field -type STRING -name productName
add-field -type STRING -name productType
add-field -type ENUM -name color -enum-values blue,green,red
add-field -type ENUM -name size -enum-values small,medium,blue
add-field -type INTEGER -name inventoryCount
## A primary key must be defined for every table
## Here, we will define field 'productName' as the primary key.
primary-key -field productName
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myProducts -wait
```

However, you can use multiple fields for your primary keys. On a functional level, doing this allows you to delete multiple rows in your table in a single atomic operation. In addition, multiple primary keys allows you to retrieve a subset of the rows in your table in a single atomic operation.

We describe how to retrieve multiple rows from your table in Reading Table Rows (page 44). We show how to delete multiple rows at a time in Using multiDelete() (page 42).

Data Type Limitations

Fields can be designated as primary keys only if they are declared to be one of the following types:

- Integer
- Long
- Float
- Double
- String
- Enum

Partial Primary Keys

Some of the methods you use to perform multi-row operations allow, or even require, a partial primary key. A partial primary key is, simply, a key where only some of the fields comprising the row's primary key are specified.

For example, the following example specifies three fields for the table's primary key:

```
## Enter into table creation mode
table create -name myProducts
## Now add the fields
add-field -type STRING -name productName
add-field -type STRING -name productType
add-field -type STRING -name productClass
add-field -type ENUM -name color -enum-values blue,green,red
add-field -type ENUM -name size -enum-values small,medium,large
add-field -type INTEGER -name inventoryCount
## A primary key must be defined for every table
primary-key -field productName -field productType -field productClass
## Exit table creation mode
exit
## Add the table to the store. Use the -wait flag to
## force the script to wait for the plan to complete
## before doing anything else.
plan add-table -name myProducts -wait
```

In this case, a full primary key would be one where you provide value for all three primary key fields: `productName`, `productType`, and `productClass`. A partial primary key would be one where you provide values for only one or two of those fields.

Note that order matters when specifying a partial key. The partial key must be a subset of the full key, starting with the first field specified and then adding fields in order. So the following partial keys are valid:

`productName`
`productName`, `productType`

But a partial key comprised of `productType` and `productClass` is not.

Shard Keys

Shard keys identify which primary key fields are meaningful in terms of shard storage. That is, rows which contain the same values for all the shard key fields are guaranteed to be stored on the same shard. This matters for some operations that promise atomicity of the results. (See Executing a Sequence of Operations (page 86) for more information.)

For example, suppose you set the following primary keys:

```
primary-key -field productType -field productName -field productClass
```

You can guarantee that rows are placed on the same shard using the values set for the `productType` and `productName` fields like this:

```
shard-key -field productType -field productName
```

Note

Shard key fields must be a first-to-last subset of the primary key fields, and they must be specified in the same order as were the primary key fields. In the previous example, the following would result in an error:

```
shard-key -field productClass
```

Row Data

There are no restrictions on the size of your rows, or the amount of data that you store in a field. However, you should consider your store's performance when deciding how large you are willing to allow your individual tables and rows to become. As is the case with any data storage scheme, the larger your rows, the longer it takes to read the information from storage, and to write the information to storage.

On the other hand, every table row carries with it some amount of overhead. Also, as the number of your rows grows very large, search times may be adversely affected. As a result, choosing to use a large number of tables, each of which use rows with just a small handful of fields, can also harm your store's performance.

Therefore, when designing your tables' content, you must find the appropriate balance between a small number of tables, each of which uses very large rows; and a large number of tables, each of which uses very small rows. You should also consider how frequently any given piece of information will be accessed.

For example, suppose your table contains information about users, where each user is identified by their first and last names (surname and familiar name). There is a set of information that you want to maintain about each user. Some of this information is small in size, and some of it is large. Some of it you expect will be frequently accessed, while other information is infrequently accessed.

Small properties are:

- name

- gender
- address
- phone number

Large properties are:

- image file
- public key 1
- public key 2
- recorded voice greeting

There are several possible ways you can organize this data. How you should do it depends on your data access patterns.

For example, suppose your application requires you to read and write all of the properties identified above every time you access a row. (This is unlikely, but it does represent the simplest case.) In that event, you might create a single table with rows containing fields for each of the properties you maintain for the users in your application.

However, the chances are good that your application will not require you to access *all* of a user's properties every time you access his information. While it is possible that you will always need to read all of the properties every time you perform a user look up, it is likely that on updates you will operate only on some properties.

Given this, it is useful to consider how frequently data will be accessed, and its size. Large, infrequently accessed properties should be placed in tables other than that used by the frequently accessed properties.

For example, for the properties identified above, suppose the application requires:

- all of the small properties to always be used whenever the user's record is accessed.
- all of the large properties to be read for simple user look ups.
- on user information updates, the public keys are always updated (written) at the same time.
- The image file and recorded voice greeting can be updated independently of everything else.

In this case, you might store user properties using a table and a child table. The parent table holds rows containing all the small properties, plus public keys. The child table contains the image file and voice greeting.

```
## Enter into table creation mode
table create -name userInfo
## Now add the fields
add-field -type STRING -name surname
```

```
add-field -type STRING -name familiarName
add-field -type STRING -name gender
add-field -type STRING -name street
add-field -type STRING -name city
add-field -type STRING -name state
add-field -type STRING -name zipcode
add-field -type STRING -name userPhone
add-field -type BINARY -name publickey1
add-field -type BINARY -name publickey2
primary-key -field surname -field familiarName
shard-key -field surname
## Exit table creation mode
exit
### Must add the parent table before we add the child
plan add-table -name userInfo -wait

table create -name userInfo.largeProps
add-field -type STRING -name propType
add-field -type BINARY -name voiceGreeting
add-field -type BINARY -name imageFile
primary-key -field propType
exit
plan add-table -name userInfo.largeProps -wait
```

Because the parent table contains all the data that is accessed whenever user data is accessed, you can update that data all at once using a single atomic operation. At the same time, you avoid retrieving the big data values whenever you retrieve a row by splitting the image data and voice greeting into a child table.

Note

You might want to consider using the key/value API for the image data and voice greeting. By doing that, you can use the Oracle NoSQL Database large object interface, which is optimized for large object support. See the *Oracle NoSQL Database Getting Started with the Key/Value API* for information on working with large objects. Note that if you use the large object interface, you can store references to the large objects (which are just strings) in your tables.

Chapter 6. Writing and Deleting Table Rows

This chapter discusses two different write operations: putting table rows into the store, and then deleting them.

Write Exceptions

There are four exceptions that you might be required to handle whenever you perform a write operation to the store. For simple cases where you use default policies, you can probably avoid explicitly handling these. However, as your code complexity increases, so too will the desirability of explicitly managing these exceptions.

The first of these is `DurabilityException`. This exception indicates that the operation cannot be completed because the durability policy cannot be met. For more information, see Durability Guarantees (page 81).

The second is `RequestTimeoutException`. This simply means that the operation could not be completed within the amount of time provided by the store's timeout property. This probably indicates an overloaded system. Perhaps your network is experiencing a slowdown, or your store's nodes are overloaded with too many operations (especially write operations) coming in too short of a period of time.

To handle a `RequestTimeoutException`, you could simply log the error and move on, or you could pause for a short period of time and then retry the operation. You could also retry the operation, but use a longer timeout value. (There is a version of the `TableAPI.put()` method that allows you to specify a timeout value for that specific operation.)

You can also receive an `IllegalArgumentException`, which will be thrown if a `Row` that you are writing to the store does not have a primary key or is otherwise invalid.

Finally, you can receive a general `FaultException`, which indicates that some exception occurred which is neither a problem with durability nor a problem with the request timeout. Your only recourse here is to either log the error and move along, or retry the operation.

Writing Rows to a Table in the Store

Writing a new row to a table in the store, and updating an existing row are usually identical operations (although methods exist that work only if the row is being updated, or only if it is being created — these are described a little later in this section).

Remember that you can only write data to a table after it has been added to the store. See Introducing Oracle NoSQL Database Tables and Indexes (page 20) for details.

To write a row to a table in the store:

1. Construct a handle for the table to which you want to write. You do this by retrieving a `TableAPI` interface instance using `KVStore.getTableAPI()`. You then use that instance to retrieve the desired table handle using `TableAPI.getTable()`. This returns a `Table` interface instance.

2. Use the `Table` instance retrieved in the previous step to create a `Row` interface instance. You use the `Table.createRow()` method to do this.
3. Write to each field in the `Row` using `Row.put()`.
4. Write the new row to the store using `TableAPI.put()`.

The following is a trivial example of writing a row to the store. It assumes that the `KVStore` handle has already been created.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Now put all of the cells in the row.
// This does NOT actually write the data to
// the store.

row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Now write the table to the store.
// "item" is the row's primary key. If we had not set that value,
// this operation will throw an IllegalArgumentException.
tableH.put(row, null, null);
```

Writing Rows to a Child Table

To write to a child table, first create the row in the parent table to which the child belongs. You do this by populating the parent row with data. Then you write the child table's row(s).

When you do, you must specify the primary key used by the parent table, as well as the primary key used by the child table's rows.

For example, in Defining Child Tables (page 27) we showed how to create a child table. To write data to that table, do this:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// First, populate a row in the parent table
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Now put all of the cells in the row.

row.put("itemCategory", "Bolts");
row.put("description", "Metric & US sizes");

// Now write the table row to the store.
tableH.put(row, null, null);

// Now populate the corresponding child table
Table myChildTable = tableH.getTable("myTable.myChildTable");

// Get a row instance
Row childRow = myChildTable.createRow();

// Populate the rows. Because the parent table's "itemCategory"
// field is a primary key, this must be populated in addition
// to all of the child table's rows
childRow.put("itemCategory", "Bolts");
childRow.put("itemSKU", "1392610");
childRow.put("itemDescription", "1/4-20 x 1/2 Grade 8 Hex");
childRow.put("price", new Float(11.99));
childRow.put("inventoryCount", 1457);
```

Other put Operations

Beyond the very simple usage of the `TableAPI.put()` method illustrated above, there are three other put operations that you can use:

- `TableAPI.putIfAbsent()`

  This method will only put the row if the row's primary key value DOES NOT currently exist in the table. That is, this method is successful only if it results in a *create* operation.

- `TableAPI.putIfPresent()`

  This method will only put the row if the row's primary key value already exists in the table. That is, this method is only successful if it results in an *update* operation.

- `TableAPI.putIfVersion()`

  This method will put the row only if the value matches the supplied version information. For more information, see Using Row Versions (page 72).

Deleting Rows from the Store

You delete a single row from the store using the `TableAPI.delete()` method. Rows are deleted based on a `PrimaryKey`, which you obtain using the `Table.createPrimaryKey()` method. You can also require a row to match a specified version before it will be deleted. To do this, use the `TableAPI.deleteIfVersion()` method. Versions are described in Using Row Versions (page 72).

When you delete a row, you must handle the same exceptions as occur when you perform any write operation on the store. See Write Exceptions (page 38) for a high-level description of these exceptions.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");
```

```
// Get the primary key for the row that we want to delete
PrimaryKey primaryKey = myTable.createPrimaryKey();
primaryKey.put("item", "Bolts");

// Delete the row
// This performs a store write operation
tableH.delete(primaryKey, null, null);
```

Using multiDelete()

You can delete multiple rows at once in a single atomic operation, so long as they all share the shard key values. Recall that shard keys are at least a subset of your primary keys. The result is that you use a partial primary key (which happens to be a shard key) to perform a multi-delete.

To delete multiple rows at once, use the `TableAPI.multiDelete()` method.

For example, suppose you created a table like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name itemType
add-field -type STRING -name itemCategory
add-field -type STRING -name itemClass
add-field -type STRING -name itemColor
add-field -type STRING -name itemSize
add-field -type FLOAT -name price
add-field -type INTEGER -name inventoryCount
primary-key -field itemType -field itemCategory -field itemClass
-field itemColor -field itemSize
shard-key -field itemType -field itemCategory -field itemClass
## Exit table creation mode
exit
```

With tables containing data like this:

- Row 1:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: small
  price: 12.07
  inventoryCount: 127

- Row 2:

  itemType: Hats

itemCategory: baseball
itemClass: longbill
itemColor: red
itemSize: medium
price: 13.07
inventoryCount: 201

- Row 3:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: large
  price: 14.07
  inventoryCount: 39

Then in this case, you can delete all the rows sharing the partial primary key `Hats`, `baseball`, `longbill` as follows:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Get the primary key for the row that we want to delete
PrimaryKey primaryKey = myTable.createPrimaryKey();
primaryKey.put("itemType", "Hats");
primaryKey.put("itemCategory", "baseball");
primaryKey.put("itemClass", "longbill");

// Exception handling omitted
tableH.multiDelete(primaryKey, null, null);
```

Chapter 7. Reading Table Rows

There are several ways to retrieve table rows from the store. You can:

1. Retrieve a single row at a time using `TableAPI.get()`.

2. Retrieve rows associated with a shard key (which is based on at least part of your primary keys) using either `TableAPI.multiGet()` or `TableAPI.multiGetIterator()`.

3. Retrieve table rows that share a shard key, or an index key, using `TableAPI.tableIterator()`.

Each of these are described in the following sections.

Read Exceptions

One of three exceptions can occur when you attempt a read operation in the store. The first of these is `ConsistencyException`. This exception indicates that the operation cannot be completed because the consistency policy cannot be met. For more information, see Consistency Guarantees (page 74).

The second exception is `RequestTimeoutException`. This means that the operation could not be completed within the amount of time provided by the store's timeout property. This probably indicates a store that is attempting to service too many read requests all at once. Remember that your data is partitioned across the shards in your store, with the partitioning occurring based on your shard keys. If you designed your keys such that a large number of read requests are occurring against a single key, you could see request timeouts even if some of the shards in your store are idle.

A request timeout could also be indicative of a network problem that is causing the network to be slow or even completely unresponsive.

To handle a `RequestTimeoutException`, you could simply log the error and move on, or you could pause for a short period of time and then retry the operation. You could also retry the operation, but use a longer timeout value.

Finally, you can receive a general `FaultException`, which indicates that some exception occurred which is neither a problem with consistency nor a problem with the request timeout. Your only recourse here is to either log the error and move along, or retry the operation.

Retrieving a Single Row

To retrieve a single row from the store:

1. Construct a handle for the table from which you want to read. You do this by retrieving a `TableAPI` class instance using `KVStore.getTableAPI()`. You then use that instance to retrieve the desired table handle using `TableAPI.getTable()`. This returns a `Table` class instance.

2. Use the `Table` instance retrieved in the previous step to create a `PrimaryKey` class instance. In this case, the key you create must be the entire primary key.

3. Retrieve the row using `TableAPI.get()`. This performs a store read operation.

4. Retrieve individual fields from the row using the `Row.get()` method.

For example, in Writing Rows to a Table in the Store (page 38) we showed a trivial example of storing a table row to the store. The following trivial example shows how to retrieve that row.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. This is driven by your table
// design, which designated one or more fields as
// being part of the table's primary key. In this
// case, we have a single field primary key, which is the
// 'item' field. Specifically, we want to retrieve the
// row where the 'item' field contains 'Bolts'.
PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
Row row = tableH.get(key, null);

// Now retrieve the individual fields from the row.
String item = row.get("item").asString.get();
String description = row.get("description").asString.get();
Integer count = row.get("count").asInteger.get();
Double percentage = row.get("percentage").asDouble.get();
```

Retrieve a Child Table

In Writing Rows to a Child Table (page 39) we showed how to populate a child table with data. To retrieve that data, you must specify the primary key used for the parent table row, as well as the primary key for the child table row. For example:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// We omit retrieval of the parent table because it is not required.
Table myChildTable = tableH.getTable("myTable.myChildTable");

// Construct the PrimaryKey. This key must contain the primary key
// from the parent table row, as well as the primary key from the
// child table row that you want to retrieve.
PrimaryKey key = myChildTable.createPrimaryKey();
key.put("itemCategory", "Bolts");
key.put("itemSKU", "1392610");

// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
Row row = tableH.get(key, null);

// Now retrieve the individual fields from the row.
String description = row.get("itemDescription").asString().get();
Float price = row.get("price").asFloat().get();
Integer invCount = row.get("inventoryCount").asInteger().get();
```

For information on how to iterate over nested tables, see Using MultiRowOptions to Retrieve Nested Tables (page 54).

Using multiGet()

`TableAPI.multiGet()` allows you to retrieve multiple rows at once, so long as they all share the same shard keys. You can specify a full set of shard keys, or a partial set.

Use `TableAPI.multiGet()` only if your retrieval set will fit entirely in memory.

For example, suppose you have a table that stores information about products, which is designed like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name itemType
add-field -type STRING -name itemCategory
add-field -type STRING -name itemClass
add-field -type STRING -name itemColor
add-field -type STRING -name itemSize
add-field -type FLOAT -name price
add-field -type INTEGER -name inventoryCount
primary-key -field itemType -field itemCategory -field itemClass
-field itemColor -field itemSize
shard-key -field itemType -field itemCategory -field itemClass
## Exit table creation mode
exit
```

With tables containing data like this:

- Row 1:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: small
  price: 12.07
  inventoryCount: 127

- Row 2:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: medium
  price: 13.07
  inventoryCount: 201

- Row 3:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: large
  price: 14.07
  inventoryCount: 39

In this case, you can retrieve all of the rows with their `itemType` field set to `Hats` and their `itemCategory` field set to `baseball`. Notice that this represents a partial primary key, because `itemClass`, `itemColor` and `itemSize` are not used for this query.

```
package kvstore.basicExample;

...

import java.util.List;
import java.util.Iterator;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.RequestTimeoutException;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
key.put("itemType", "Hats");
key.put("itemCategory", "baseball");

List<Row> myRows = null;

try {
    myRows = tableH.multiGet(key, null, null);
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
} catch (RequestTimeoutException re) {
    // The operation was not completed within the
    // timeout value
}
```

You can then iterate over the resulting list as follows:

```
for (Rows theRow: myRows) {
    String itemType = theRow.get("itemType").asString.get();
    String itemCategory = theRow.get("itemCategory").asString.get();
    String itemClass = theRow.get("itemClass").asString.get();
    String itemColor = theRow.get("itemColor").asString.get();
    String itemSize = theRow.get("itemSize").asString.get();
    Float price = theRow.get("price").asFloat.get();
    Integer price = theRow.get("itemCount").asInteger.get();
}
```

Iterating over Table Rows

`TableAPI.tableIterator()` provides non-atomic table iteration. Use this method to iterate over indexes. This method performs a parallel scan of your tables if you set a concurrent request size other than 1.

`TableAPI.tableIterator()` does not return the entire set of rows all at once. Instead, it batches the fetching of rows in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth. Also, the rows returned by this method are in unsorted order.

Note that this method does not result in a single atomic operation. Because the retrieval is batched, the return set can change over the course of the entire retrieval operation. As a result, you lose the atomicity of the operation when you use this method.

This method provides for an unsorted traversal of rows in your table. If you do not provide a key, then this method will iterate over all of the table's rows.

When using this method, you can optionally specify:

- A `MultiRowOptions` class instance. This class allows you to specify a field range, and the ancestor and parent tables you want to include in this iteration.
- A `TableIteratorOptions` class instance. This class allows you to identify the suggested number of keys to fetch during each network round trip. If you provide a value of 0, an internally determined default is used. You can also use this class to specify the traversal order (`FORWARD` and `UNORDERED` are the only options).

  This class also allows you to control how many threads are used to perform the store read. By default this method determines the degree of concurrency based on the number of available processors. You can tune this concurrency by explicitly stating how many threads to use for table retrieval. See Parallel Scans (page 61) for more information.

  Finally, you use this class to specify a consistency policy. See Consistency Guarantees (page 74) for more information.

Note

When using `TableAPI.tableIterator()`, it is important to call `TableIterator.close()` when you are done with the iterator to avoid resource

leaks. This is especially true for long-running applications, especially if you do not iterate over the entire result set.

For example, suppose you have a table that stores information about products, which is designed like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name itemType
add-field -type STRING -name itemCategory
add-field -type STRING -name itemClass
add-field -type STRING -name itemColor
add-field -type STRING -name itemSize
add-field -type FLOAT -name price
add-field -type INTEGER -name inventoryCount
primary-key -field itemType -field itemCategory -field itemClass
-field itemColor -field itemSize
shard-key -field itemType -field itemCategory -field itemClass
## Exit table creation mode
exit
```

With tables containing data like this:

- Row 1:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: small
  price: 12.07
  inventoryCount: 127

- Row 2:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: medium
  price: 13.07
  inventoryCount: 201

- Row 3:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: large

price: 14.07
inventoryCount: 39

- Row *n*:

itemType: Coats
itemCategory: Casual
itemClass: Winter
itemColor: red
itemSize: large
price: 247.99
inventoryCount: 9

Then in the simplest case, you can retrieve all of the rows related to 'Hats' using `TableAPI.TableIterator` as follows. Note that this simple example can also be accomplished with `TableAPI.multiGet()`. If you have a complete shard key, and if the entire results set will fit in memory, then `TableAPI.multiGet()` will perform much better than `TableAPI.TableIterator`. However, if the results set cannot fit entirely in memory, or if you do not have a complete shard key, then `TableAPI.TableIterator` is the better choice. Note that reads performed using `TableAPI.TableIterator` are non-atomic, which may have ramifications if you are performing a long-running iteration over records that are being updated..

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
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;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
```

```
key.put("itemType", "Hats");

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Specifying Field Ranges

When performing multi-key operations in the store, you can specify a range of rows to operate upon. You do this using the `FieldRange` class, which is accepted by any of the methods which perform bulk reads. This class is used to restrict the selected rows to those matching a range of field values.

For example, suppose you defined a table like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name surname
add-field -type STRING -name familiarName
add-field -type STRING -name userID
add-field -type STRING -name phonenumber
add-field -type STRING -name address
add-field -type STRING -name email
add-field -type STRING -name dateOfBirth
primary-key -field surname -field familiarName -field userID
shard-key -field surname -field familiarName
## Exit table creation mode
exit
```

The `surname` contains a person's family name, such as `Smith`. The `familiarName` contains their common name, such as `Bob`, `Patricia`, `Robert`, and so forth.

Given this, you could perform operations for all the rows related to users named `Bob Smith` and `Patricia Smith` by specifying a field range.

A `FieldRange` class instance is created using `Table.createFieldRange()`. This method takes just one argument — the name of the primary key for which you want to set the range. Once you have the `FieldRange` object, you can set the start and end values for the range. You also indicate whether the range values are inclusive.

In this case, we will define the start of the key range using the string "Bob" and the end of the key range to be "Patricia". Both ends of the key range will be inclusive.

In this example, we use `TableIterator`, but we could just as easily use this range on any multi-row read operation, such as `TableAPI.multiGet()` or `TableAPI.multiGetKeys()`. The `FieldRange` object is passed to these methods using a `MultiRowOptions` class instance, which we construct using the `FieldRange.createMultiRowOptions()` convenience method.

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.MultiRowOptions;
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;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
key.put("surname", "Smith");

// Create the field range.
FieldRange fh = myTable.createFieldRange("familiarName");
fh.setStart("Bob", true);
fh.setEnd("Patricia", true);
MultiRowOptions mro = fh.createMultiRowOptions();

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
try {
```

```
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Using MultiRowOptions to Retrieve Nested Tables

When you are iterating over a table, or performing a multi-get operation, by default only rows are retrieved from the table on which you are operating. However, you can use `MultiRowOptions` to specify that parent and child tables are to be retrieved as well.

When you do this, parent tables are retrieved first, then the table you are operating on, then child tables. In other words, the tables' hierarchical order is observed.

The parent and child tables retrieved are identified by specifying a `List` of `Table` objects to the `ancestors` and `children` parameters on the class constructor. You can also specify these using the `MultiRowOptions.setIncludedChildTables()` or `MultiRowOptions.setIncludedParentTables()` methods.

When operating on rows retrieved from multiple tables, it is your responsibility to determine which table the row belongs to.

For example, suppose you create a table with a child and grandchild table like this:

```
table create -name prodTable
add-field -type STRING -name prodType
add-field -type STRING -name typeDescription
primary-key -field prodType
exit
plan add-table -name prodTable -wait

table create -name prodTable.prodCategory
add-field -type STRING -name categoryName
add-field -type STRING -name categoryDescription
primary-key -field categoryName
exit
plan add-table -name prodTable.prodCategory -wait

table create -name prodTable.prodCategory.item
add-field -type STRING -name itemSKU
add-field -type STRING -name itemDescription
add-field -type FLOAT -name itemPrice
add-field -type STRING -name vendorUID
add-field -type INTEGER -name inventoryCount
primary-key -field itemSKU
```

```
exit
plan add-table -name prodTable.prodCategory.item -wait
```

With tables containing data like this:

- Row 1:

  prodType: Hardware
  typeDescription: Equipment, tools and parts

  - Row 1.1:

    categoryName: Bolts
    categoryDescription: Metric & US Sizes

    - Row 1.1.1:

      itemSKU: 1392610
      itemDescription: 1/4-20 x 1/2 Grade 8 Hex
      itemPrice: 11.99
      vendorUID: A8LN99
      inventoryCount: 1457

- Row 2:

  prodType: Tools
  typeDescription: Hand and power tools

  - Row 2.1:

    categoryName: Handtools
    categoryDescription: Hammers, screwdrivers, saws

    - Row 2.1.1:

      itemSKU: 1582178
      itemDescription: Acme 20 ounce claw hammer
      itemPrice: 24.98
      vendorUID: D6BQ27
      inventoryCount: 249

In this case, you can display all of the data contained in these tables in the following way.

Start by getting all our table handles:

```
package kvstore.tableExample;

import java.util.Arrays;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
```

```
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

import oracle.kv.table.TableIterator;
import oracle.kv.table.MultiRowOptions;

...

private static Table prodTable;
private static Table categoryTable;
private static Table itemTable;

private static TableAPI tableH;

...

// KVStore handle creation is omitted for brevity

...

tableH = kvstore.getTableAPI();
prodTable = tableH.getTable("prodTable");
categoryTable = tableH.getTable("prodTable.prodCategory");
itemTable = tableH.getTable("prodTable.prodCategory.item");
```

Now we need the `PrimaryKey` and the `MultiRowOptions` that we will use to iterate over the top-level table. Because we want all the rows in the top-level table, we create an empty `PrimaryKey`.

The `MultiRowOptions` identifies the two child tables in the constructor's `child` parameter. This causes the iteration to return all the rows from the top-level table, as well as all the rows from the nested children tables.

```
// Construct a primary key
PrimaryKey key = prodTable.createPrimaryKey();

// Get a MultiRowOptions and tell it to look at both the child
// tables
MultiRowOptions mro = new MultiRowOptions(null, null,
        Arrays.asList(categoryTable, itemTable));
```

Now we perform the iteration:

```
// Get the table iterator
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
```

```
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        displayRow(row);
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Our `displayRow()` method is used to determine which table a row belongs to, and then display it in the appropriate way.

```
private static void displayRow(Row row) {
    // Display the row depending on which table it belongs to
    if (row.getTable().equals(prodTable)) {
        displayProdTableRow(row);
    } else if (row.getTable().equals(categoryTable)) {
        displayCategoryTableRow(row);
    } else {
        displayItemTableRow(row);
    }
}
```

Finally, we just need the methods used to display each row. These are trivial, but in a more sophisticated application they could be used to do more complex things, such as construct HTML pages or write XSL-FO for the purposes of generating PDF copies of a report.

```
private static void displayProdTableRow(Row row) {
    System.out.println("\nType: " +
        row.get("prodType").asString().get());
    System.out.println("Description: " +
        row.get("typeDescription").asString().get());
}

private static void displayCategoryTableRow(Row row) {
    System.out.println("\tCategory: " +
        row.get("categoryName").asString().get());
    System.out.println("\tDescription: " +
        row.get("categoryDescription").asString().get());
}

private static void displayItemTableRow(Row row) {
    System.out.println("\t\tSKU: " +
        row.get("itemSKU").asString().get());
    System.out.println("\t\tDescription: " +
        row.get("itemDescription").asString().get());
    System.out.println("\t\tPrice: " +
        row.get("itemPrice").asFloat().get());
    System.out.println("\t\tVendorUID: " +
```

```
            row.get("vendorUID").asString().get());
        System.out.println("\t\tInventory count: " +
            row.get("inventoryCount").asInteger().get());
        System.out.println("\n");
    }
```

Note that the retrieval order remains the top-most ancestor to the lowest child, even if you retrieve by lowest child. For example, you can retrieve all the Bolts, and all of their parent tables, like this:

```
// Get all the table handles
prodTable = tableH.getTable("prodTable");
categoryTable = tableH.getTable("prodTable.prodCategory");
itemTable = tableH.getTable("prodTable.prodCategory.item");

// Construct a primary key
PrimaryKey key = itemTable.createPrimaryKey();
key.put("prodType", "Hardware");
key.put("categoryName", "Bolts");

// Get a MultiRowOptions and tell it to look at both the ancestor
// tables
MultiRowOptions mro = new MultiRowOptions(null,
        Arrays.asList(prodTable, categoryTable), null);

// Get the table iterator
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        displayRow(row);
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Reading Indexes

You use `TableIterator` to retrieve table rows using a table's indexes. Just as when you use `TableIterator` to read table rows using a table's primary key(s), when reading using indexes you can set options such as field ranges, traversal direction, and so forth. By default, index scans return entries in order (`Direction.FORWARD`).

In this case, rather than provide `TableIterator` with a `PrimaryKey` instance, you use an instance of `IndexKey`.

For example, suppose you defined a table like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name surname
add-field -type STRING -name familiarName
add-field -type STRING -name userID
add-field -type STRING -name phonenumber
add-field -type STRING -name address
add-field -type STRING -name email
add-field -type STRING -name dateOfBirth
primary-key -field surname -field familiarName -field userID
shard-key -field surname -field familiarName
## Exit table creation mode
exit
plan add-table -name myTable -wait
plan add-index -name DoB -table myTable -field dateOfBirth -wait
```

This creates an index named `DoB` for table `myTable` based on the value of the `dateOfBirth` field. To read using that index, you use `Table.getIndex()` to retrieve the index named `Dob`. You then create an `IndexKey` from the `Index` object. For example:

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

// Exception handling is omitted, but in production code
```

```
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

In the previous example, the code examines every row indexed by the `DoB` index. A more likely, and useful, example in this case would be to limit the rows returned through the use of a field range. You do that by using `Index.createFieldRange()` to create a `FieldRange` object. At that time, you must specify the field to base the range on. Recall that an index can be based on more than one table field, so the field name you give the method must be one of the indexed fields.

For example, if the rows hold dates in the form of `yyyy-mm-dd`, you could retrieve all the people born in the month of May, 1994 in the following way. This index only examines one field, `dateOfBirth`, so we give that field name to `Index.createFieldRange()`:

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.MultiRowOption;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the IndexKey. The name we gave our index when
```

```
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

// Create the field range.
FieldRange fh = dobIdx.createFieldRange("dateOfBirth");
fh.setStart("1994-05-01", true);
fh.setEnd("1994-05-30", true);
MultiRowOptions mro = fh.createMultiRowOptions();

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Parallel Scans

By default, store reads are performed using multiple threads, the number of which is chosen by the number of cores available to your code. You can configure the maximum number of client-side threads to be used for the scan, as well as the number of results per request and the maximum number of result batches that the Oracle NoSQL Database client can hold before the scan pauses. To do this, use the `TableIteratorOptions` class. You pass this to `TableAPI.tableIterator()`. This creates a `TableIterator` that uses the specified parallel scan configuration.

Note

You cannot configure the number of scans you use for your reads if you are using indexes.

For example, to retrieve all of the records in the store using 5 threads in parallel, you would do this:

```
package kvstore.basicExample;

...

import oracle.kv.Consistency;
import oracle.kv.Direction;
import oracle.kv.KVStore;
```

```
import oracle.kv.table.FieldRange;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.MultiRowOption;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.TableIteratorOptions;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey.
PrimaryKey key = myTable.createPrimaryKey();
key.put("itemType", "Hats");
key.put("itemCategory", "baseball");

TableIteratorOptions tio =
    new TableIteratorOptions(Direction.UNORDERED,
                             Consistency.NONE_REQUIRED,
                             0,     // timeout
                             null,  // timeout units
                             5,     // number of concurrent
                                    // threads
                             0,     // results per request
                             0);    // max result sets
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter =
    tableH.tableIterator(key, null, tio);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Chapter 8. Using Data Types

Many of the types that Oracle NoSQL Database offers are easy to use. Examples of their usage has been scattered throughout this manual. However, some types are a little more complicated to use because they use container methods. They are also declared a little bit differently when defining tables using the CLI. This chapter describes their usage.

The types described in this chapter are: Arrays, Maps, Records, Enums, and Byte Arrays.

This chapter shows how to read and write values of each of these types.

Using Arrays

Arrays are a sequence of values all of the same type.

When you declare a table field as an array, you use `add-array-field`. This puts you into a special submode where you add one and only one field. The name you give the field is optional, unimportant and will be ignored. What you are really doing when you add this sub-field is declaring the type of the elements in the array.

To define a simple two-column table where the primary key is a UID and the second column contains array of strings, you use the following script:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type INTEGER -name uid

## Create an array field. This puts us into a new submode.
add-array-field -name myArray
## Sets the type for the array. The name is required but ignored
add-field -type STRING -name arrayField

### Exit array field creation mode
exit

## A primary key must be defined for every table
primary-key -field uid

## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the array, use `Row.putArray()`, which returns an `ArrayValue` class instance. You then use `ArrayValue.put()` to write elements to the array:

```
TableAPI tableH = kvstore.getTableAPI();
```

```
Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

ArrayValue av = row.putArray("myArray");
av.add("One");
av.add("Two");
av.add("Three");

tableH.put(row, null, null);
```

Note that `ArrayValue` has methods that allow you to add multiple values to the array by appending an array of values to the array. This assumes the array of values matches the array's schema. For example, the previous example could be done in the following way:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

ArrayValue av = row.putArray("myArray");
String myStrings[] = {"One", "Two", "Three"};
av.add(myStrings);

tableH.put(row, null, null);
```

To read the array, use `Row.get().asArray()`. This returns an `ArrayValue` class instance. You can then use `ArrayValue.get()` to retrieve an element of the array from a specified index, or you can use `ArrayValue.toList()` to return the array as a Java `List`. In either case, the retrieved values are returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asString()`.

For example, to iterate over the array created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

/* Iterate over the array, displaying each element as a string */
ArrayValue av = row.get("myArray").asArray();
for (FieldValue fv: av.toList()) {
    System.out.println(fv.asString().get()); }
```

Using Binary

You can declare a field as BINARY using `add-field -type BINARY`. You then read and write the field value using a Java byte array.

If you want to store a large binary object, then you should use the LOB APIs rather than a BINARY field. For information on using the LOB APIs, see the Oracle NoSQL API Large Object API introduction.

Note that FIXED_BINARY should be used over the BINARY datatype any time you know that all the field values will be of the same size. FIXED_BINARY is a more compact storage format because it does not need to store the size of the array. See Using Fixed Binary (page 67) for information on the FIXED_BINARY datatype.

To define a simple two-column table where the primary key is a UID and the second column contains a binary field, you use the following script:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type INTEGER -name uid

## Create an enum field
add-field -type BINARY -name myBinary

## A primary key must be defined for every table
primary-key -field uid

## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the byte array, use `Row.put()`.

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

String aString = "The quick brown fox.";
try {
    row.put("myByteArray", aString.getBytes("UTF-8"));
} catch (UnsupportedEncodingException uee) {
    uee.printStackTrace();
}

tableH.put(row, null, null);
```

To read the binary field, use `Row.get().asBinary()`. This returns a `BinaryValue` class instance. You can then use `BinaryValue.get()` to retrieve the stored byte array.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

byte[] b = row.get("myByteArray").asBinary().get();
String aString = new String(b);
System.out.println("aString: " + aString);
```

Using Enums

When you declare a table field as an enum, you use `add-field -type ENUM`. You must also use the `-enum-values` parameter to declare the acceptable enumeration values.

To define a simple two-column table where the primary key is a UID and the second column contains an enum, you use the following script:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type INTEGER -name uid

## Create an enum field
add-field -type ENUM -name myEnum -enum-values Apple,Pears,Oranges

## A primary key must be defined for every table
primary-key -field uid

## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the enum, use `Row.putEnum()`. If the enumeration value that you use with this method does not match a value defined on the `-enum-values` parameter during table definition, an `IllegalArgumentException` is thrown.

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");
```

```
Row row = myTable.createRow();
row.put("uid", 12345);

row.putEnum("myEnum", "Pears");

tableH.put(row, null, null);
```

To read the enum, use `Row.get().asEnum()`. This returns a `EnumValue` class instance. You can then use `EnumValue.get()` to retrieve the stored enum value's name as a string. Alternatively, you can use `EnumValue.getIndex()` to retrieve the stored value's index position.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

EnumValue ev = row.get("testEnum").asEnum();
System.out.println("enum as string: " +
                ev.get()); // returns "Pears"
System.out.println("enum index: " +
                ev.getIndex()); // returns '1'
```

Using Fixed Binary

You can declare a field as FIXED_BINARY using `add-field -type FIXED_BINARY`. When you do this, you must also specify the field's size using the `-size` parameter. You then read and write the field value using Java byte arrays. However, if the byte array does not equal the specified size, then `IllegalArgumentException` is thrown when you attempt to write the field. Write the field value using a Java byte array.

If you want to store a large binary object, then you should use the LOB APIs rather than a BINARY field. For information on using the LOB APIs, see the Oracle NoSQL API Large Object API introduction.

FIXED_BINARY should be used over the BINARY datatype any time you know that all the field values will be of the same size. FIXED_BINARY is a more compact storage format because it does not need to store the size of the array. See Using Binary (page 65) for information on the BINARY datatype.

To define a simple two-column table where the primary key is a UID and the second column contains a fixed binary field, you use the following script:

```
## Enter into table creation mode
table create -name myTable
```

```
## Now add the fields
add-field -type INTEGER -name uid

## Create an enum field
add-field -type FIXED_BINARY -size 20 -name myBinary

## A primary key must be defined for every table
primary-key -field uid

## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the byte array, use `Row.putFixed()`. Again, if the byte array does not match the size defined for this field, then `IllegalArgumentException` is thrown.

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

String aString = "The quick brown fox.";
try {
    row.putFixed("myByteArray", aString.getBytes("UTF-8"));
} catch (UnsupportedEncodingException uee) {
    uee.printStackTrace();
}

tableH.put(row, null, null);
```

To read the fixed binary field, use `Row.get().asFixedBinary()`. This returns a `FixedBinaryValue` class instance. You can then use `FixedBinaryValue.get()` to retrieve the stored byte array.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

byte[] b = row.get("myByteArray").asFixedBinary().get();
String aString = new String(b);
```

```
System.out.println("aString: " + aString);
```

Using Maps

All map entries must be of the same type. Regardless of the type of the map's values, its keys are always strings.

When you declare a table field as a map, you use `add-map-field`. This puts you into a special submode where you add one and only one field. The name you give the field is unimportant and will be ignored. What you are really doing when you add this sub-field is declaring the type of the elements in the map.

To define a simple two-column table where the primary key is a UID and the second column contains a map of integers, you use the following script:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type INTEGER -name uid

## Create a map field. This puts us into a new submode.
add-map-field -name myMap
## Sets the type for the map. The name is required but ignored
add-field -type INTEGER -name mapField

### Exit array map creation mode
exit

## A primary key must be defined for every table
primary-key -field uid

## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the map, use `Row.putMap()`, which returns a `MapValue` class instance. You then use `MapValue.put()` to write elements to the map:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

MapValue mv = row.putMap("myMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
```

```
tableH.put(row, null, null);
```

To read the map, use `Row.get().asMap()`. This returns a `MapValue` class instance. You can then use `MapValue.get()` to retrieve an map value. The retrieved value is returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asInteger()`.

For example, to retrieve elements from the map created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

MapValue mv = row.get("testMap").asMap();
FieldValue fv = mv.get("field3");
System.out.println("fv: " + fv.asInteger().get());
```

Using Embedded Records

A record entry can contain fields of differing types. However, embedded records should be used only when the data is relatively static. In general, child tables provide a better solution over embedded records, especially if the child dataset is large or is likely to change in size.

When you declare a table field as a record, you use `add-record-field`. This puts you into a special submode that allows you to identify the record's various fields.

To define a simple two-column table where the primary key is a UID and the second column contains a record, you use the following script:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type INTEGER -name uid

## Create a record field. This puts us into a new submode.
add-record-field -name myRecord
add-field -type STRING -name firstField
add-field -type INTEGER -name secondField

### Exit array record creation mode
exit

## A primary key must be defined for every table
primary-key -field uid
```

```
## Exit table creation mode
exit

## Add the table to the store.
plan add-table -name myTable -wait
```

To write the record, use `Row.putRecord()`, which returns a `RecordValue` class instance. You then use `RecordValue.put()` to write fields to the record:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

Row row = myTable.createRow();
row.put("uid", "12345");
RecordValue rv = row.putRecord("myRecord");
rv.put("firstField", "An embedded record STRING field");
rv.put("secondField", 3388);

tableH.put(row, null, null);
```

To read the record, use `Row.get().asRecord()`. This returns a `RecordValue` class instance. You can then use `RecordValue.get()` to retrieve a field from the record. The retrieved value is returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asInteger()`.

For example, to retrieve field values from the embedded record created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

RecordValue rv = row.get("myRecord").asRecord();
FieldValue fv = rv.get("firstField");
System.out.println("firstField: " + fv.asString().get());
fv = rv.get("secondField");
System.out.println("secondField: " + fv.asInteger().get());
```

Chapter 9. Using Row Versions

When a row is initially inserted in the store, and each time it is updated, it is assigned a unique version token. The version is always returned by the method that wrote to the store (for example, `TableAPI.put()`). The version information is also returned by methods that retrieve rows from the store.

There are two reasons why versions might be important.

1. When an update or delete is to be performed, it may be important to only perform the operation if the row's value has not changed. This is particularly interesting in an application where there can be multiple threads or processes simultaneously operating on the row. In this case, read the row, examining its version when you do so. You can then perform a put operation, but only allow the put to proceed if the version has not changed (this is often referred to as a *Compare and Set* (CAS) or *Read, Modify, Write* (RMW) operation). You use `TableAPI.putIfVersion()` or `TableAPI.deleteIfVersion()` to guarantee this.

2. When a client reads data that was previously written, it may be important to ensure that the Oracle NoSQL Database node servicing the read operation has been updated with the information previously written. This can be accomplished by passing the version of the previously written data as a consistency parameter to the read operation. For more information on using consistency, see Consistency Guarantees (page 74).

Versions are managed using the `Version` class. In some situations, it is returned as part of another encapsulating class, such as the `Row` class.

The following code fragment retrieves a row, and then stores that row only if the version has not changed:

```
package kvstore.basicExample;

...

import oracle.kv.Version;
import oracle.kv.KVStore;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// Retrieve the row. Note that we do not show the creation of
// the kvstore handle here.

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");
```

```
// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

TableIterator<Row> iter =
    tableH.tableIterator(dobIdxKey, null, null);

while (iter.hasNext()) {
    Row aRow = iter.next();

    // Retrieve the row's version information
    Version rowVersion = aRow.getVersion();

    //////////////////////////
    // Do work on the row here
    //////////////////////////

    // Put if the version is correct. Notice that here we examine
    // the return code. If it is null, that means that the put was
    // unsuccessful, probably because the row was changed elsewhere.

    Version newVersion =
        tableH.putIfVersion(row, rowVersion, null, null);
    if (newVersion == null) {
        // Unsuccessful. Someone else probably modified the record.
    }
}
```

Chapter 10. Consistency Guarantees

The KV store is built from some number of computers (generically referred to as *nodes*) that are working together using a network. All data in your store is first written to a master node. The master node then copies that data to other nodes in the store. Nodes which are not master nodes are referred to as *replicas*.

Because of the relatively slow performance of distributed systems, there can be a possibility that, at any given moment, a write operation that was performed on the master node will not yet have been performed on some other node in the store.

Consistency, then, is the policy describing whether it is possible for a row on Node A to be different from the same row on Node B.

When there is a high likelihood that a row stored on one node is identical to the same row stored on another node, we say that we have a *high consistency guarantee*. Likewise, a *low consistency guarantee* means that there is a good possibility that a row on one node differs in some way from the same row stored on another node.

You can control how high you want your consistency guarantee to be. Note that the trade-off in setting a high consistency guarantee is that your store's read performance might not be as high as if you use a low consistency guarantee.

There are several different forms of consistency guarantees that you can use. They are described in the following sections.

Note that by default, Oracle NoSQL Database uses the lowest consistency guarantee possible.

Specifying Consistency Policies

To specify a consistency policy, you use one of the static instances of the `Consistency` class, or one of its nested classes.

Once you have selected a consistency policy, you can put it to use in one of two ways. First, you can use it to define a default consistency policy using the `KVStoreConfig.setConsistency()` method. Use of this method means that all store operations will use that policy, unless they are overridden on an operation by operation basis.

The second way to use a consistency policy is to override the default policy that you are using to perform the store operation.

The following example shows how to set a default consistency policy for the store. We will show the per-operation usage of the `Consistency` class in the following sections.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
```

```
...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

kconfig.setConsistency(Consistency.NONE_REQUIRED);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

Using Predefined Consistency

You can use static instances of the `Consistency` base class to specify certain rigid consistency guarantees. There are two such instances that you can use:

1. `Consistency.ABSOLUTE`

   Requires that the operation be serviced at the master node. In this way, the row(s) will always be consistent with the master.

   This is the strongest possible consistency guarantee that you can require, but it comes at the cost of servicing all read and write requests at the master node. If you direct all your traffic to the master node (which is just one machine for each partition), then you will not be distributing your read operations across your replicas. You also will slow your write operations because your master will be busy servicing read requests. For this reason, you should use this consistency guarantee sparingly.

2. `Consistency.NONE_REQUIRED`

   Allows the store operation to proceed regardless of the state of the replica relative to the master. This is the most relaxed consistency guarantee that you can require. It allows for the maximum possible store performance, but at the high possibility that your application will be operating on stale or out-of-date information.

3. `Consistency.NONE_REQUIRED_NO_MASTER`

   Requires read operations to be serviced on a replica; never the Master. When this policy is used, the read operation will not be performed if the only node available is the Master.

   Where possible, this consistency policy should be avoided in favor of the secondary zones feature.

For example, suppose you are performing a critical read operation that you know must absolutely have the most up-to-date data. Then do this:

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.ReadOptions;
```

```
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey.
PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Create the ReadOption with our Consistency policy
ReadOptions ro = new ReadOptions(Consistency.ABSOLUTE,
                                 0,     // Timeout parameter.
                                        // 0 means use the default.
                                 null); // Timeout units. Null because
                                        // the Timeout is 0.


// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
try {
    Row row = tableH.get(key, ro);
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
}
```

Using Time-Based Consistency

A time-based consistency policy describes the amount of time that a replica node is allowed to lag behind the master node. If the replica's data is more than the specified amount of time out-of-date relative to the master, then a `ConsistencyException` is thrown. In that event, you can either abandon the operation, retry it immediately, or pause and then retry it.

In order for this type of a consistency policy to be effective, the clocks on all the nodes in the store must be synchronized using a protocol such as NTP.

In order to specify a time-based consistency policy, you use the `Consistency.Time` class. The constructor for this class requires the following information:

- `permissibleLag`

  A `long` that describes the number of `TimeUnits` the replica is allowed to lag behind the master.

- `permissibleLagUnits`

  A `TimeUnit` that identifies the units used by `permissibleLag`. For example: `TimeUnit.MILLISECONDS`.

- `timeout`

  A `long` that describes how long the replica is permitted to wait in an attempt to meet the `permissibleLag` limit. That is, if the replica cannot immediately meet the `permissibleLag` requirement, then it will wait this amount of time to see if it is updated with the required data from the master. If the replica cannot meet the `permissibleLag` requirement within the `timeout` period, a `ConsistencyException` is thrown.

- `timeoutUnit`

  A `TimeUnit` that identifies the units used by `timeout`. For example: `TimeUnit.SECONDS`.

The following sets a default time-based consistency policy of 2 seconds. The timeout is 4 seconds.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import java.util.concurrent.TimeUnit;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

Consistency.Time cpolicy =
        new Consistency.Time(2, TimeUnit.SECONDS,
                             4, TimeUnit.SECONDS);
kconfig.setConsistency(cpolicy);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

Using Version-Based Consistency

Version-based consistency is used on a per-operation basis. It ensures that a read performed on a replica is at least as current as some previous write performed on the master.

An example of how this might be used is a web application that collects some information from a customer (such as her name). It then customizes all subsequent pages presented to the customer with her name. The storage of the customer's name is a write operation that can only be performed by the master node, while subsequent page creation is performed as a read-only operation that can occur at any node in the store.

Use of this consistency policy might require that version information be transferred between processes in your application.

To create a version-based consistency policy, you use the `Consistency.Version` class. When you construct an instance of this class, you must provide the following information:

- `version`

  The `Version` that the read must match.

- `timeout`

  A `long` that describes how long the replica is permitted to wait in an attempt to meet the `version` requirement. That is, if the replica cannot immediately meet the `version` requirement, then it will wait this amount of time to see if it is updated with the required data from the master. If the replica cannot meet the requirement within the `timeout` period, a `ConsistencyException` is thrown.

- `timeoutUnit`

  A `TimeUnit` that identifies the units used by `timeout`. For example: `TimeUnit.SECONDS`.

For example, the following code performs a store write, collects the version information, then uses it to construct a version-based consistency policy.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.Version;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();
```

```
// Now put all of the cells in the row.

row.put("item", "Bolts");
row.put("count1", 5);
row.put("count2", 23);
row.put("percentage", 0.2173913);

// Now write the table to the store, capturing the
// Version information as we do.

Version matchVersion = tableH.put(row, null, null);

Version matchVersion = kvstore.put(myKey, myValue);
```

At some other point in this application's code, or perhaps in another application entirely, we use the `matchVersion` captured above to create a version-based consistency policy.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.ReadOptions;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

import java.util.concurrent.TimeUnit;

...

// KVStore handle creation is omitted for brevity

...

// Construct the PrimaryKey.

PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Create the consistency policy, using the
// Version object we captured, above.
Consistency.Version versionConsistency =
        new Consistency.Version(matchVersion,
                                200,
                                TimeUnit.NANOSECONDS);

// Create a ReadOptions using our new consistency policy.
ReadOptions ro = new ReadOptions(versionConsistency, 0, null);
```

```
// Now perform the read.
try {

    Row row = tableH.get(key, ro);

    // Do work with the row here
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
}
```

Chapter 11. Durability Guarantees

Writes are performed in the KV store by performing the write operation (be it a creation, update, or delete operation) on a master node. As a part of performing the write operation, the master node will usually make sure that the operation has made it to stable storage before considering the operation complete.

The master node will also transmit the write operation to the replica nodes in its shard. It is possible to ask the master node to wait for acknowledgments from its replicas before considering the operation complete.

Note

If your store is configured such that secondary zones are in use, then write acknowledgements are never required for the replicas in the secondary zones. That is, write acknowledgements are only returned by replicas in primary zones. See the *Oracle NoSQL Database Administrator's Guide* for more information on zones.

The replicas, in turn, will not acknowledge the write operation until they have applied the operation to their own database.

A *durability guarantee,* then, is a policy which describes how strongly persistent your data is in the event of some kind of catastrophic failure within the store. (Examples of a catastrophic failure are power outages, disk crashes, physical memory corruption, or even fatal application programming errors.)

A high durability guarantee means that there is a very high probability that the write operation will be retained in the event of a catastrophic failure. A low durability guarantee means that the write is very unlikely to be retained in the event of a catastrophic failure.

The higher your durability guarantee, the slower your write-throughput will be in the store. This is because a high durability guarantee requires a great deal of disk and network activity.

Usually you want some kind of a durability guarantee, although if you have highly transient data that changes from run-time to run-time, you might want the lowest possible durability guarantee for that data.

Durability guarantees include two types of information: acknowledgment guarantees and synchronization guarantees. These two types of guarantees are described in the next sections. We then show how to set a durability guarantee.

Note that by default, Oracle NoSQL Database uses a low durability guarantee.

Setting Acknowledgment-Based Durability Policies

Whenever a master node performs a write operation (create, update or delete), it must send that operation to its various replica nodes. The replica nodes then apply the write operation(s) to their local databases so that the replicas are consistent relative to the master node.

Upon successfully applying write operations to their local databases, replicas in primary zones send an *acknowledgment message* back to the master node. This message simply says that the write operation was received and successfully applied to the replica's local database. Replicas in secondary zones do not send these acknowledgement messages.

Note

The exception to this are replicas in secondary zones, which will never acknowledge write operations. See the *Oracle NoSQL Database Administrator's Guide* for more information on zones.

An acknowledgment-based durability policy describes whether the master node will wait for these acknowledgments before considering the write operation to have completed successfully. You can require the master node to wait for no acknowledgments, acknowledgments from a simple majority of replica nodes in primary zones, or acknowledgments from all replica nodes in primary zones.

The more acknowledgments the master requires, the slower its write performance will be. Waiting for acknowledgments means waiting for a write message to travel from the master to the replicas, then for the write operation to be performed at the replica (this may mean disk I/O), then for an acknowledgment message to travel from the replica back to the master. From a computer application's point of view, this can all take a long time.

When setting an acknowledgment-based durability policy, you can require acknowledgment from:

- All replicas. That is, all of the replica nodes in the shard that reside in a primary zone. Remember that your store has more than one shard, so the master node is not waiting for acknowledgments from every machine in the store.
- No replicas. In this case, the master returns with normal status from the write operation as soon as it has met its synchronization-based durability policy. These are described in the next section.
- A simple majority of replicas in primary zones. That is, if the shard has 5 replica nodes residing in primary zones, then the master will wait for acknowledgments from 3 nodes.

Setting Synchronization-Based Durability Policies

Whenever a node performs a write operation, the node must know whether it should wait for the data to be written to stable storage before successfully returning from the operation.

As a part of performing a write operation, the data modification is first made to an in-memory cache. It is then written to the filesystem's data buffers. And, finally, the contents of the data buffers are synchronized to stable storage (typically, a hard drive).

You can control how much of this process the master node will wait to complete before it returns from the write operation with a normal status. There are three different levels of synchronization durability that you can require:

- NO_SYNC

The data is written to the host's in-memory cache, but the master node does not wait for the data to be written to the file system's data buffers, or for the data to be physically transferred to stable storage. This is the fastest, but least durable, synchronization policy.

- WRITE_NO_SYNC

  The data is written to the in-memory cache, and then written to the file system's data buffers, but the data is not necessarily transferred to stable storage before the operation completes normally.

- SYNC

  The data is written to the in-memory cache, then transferred to the file system's data buffers, and then synchronized to stable storage before the write operation completes normally. This is the slowest, but most durable, synchronization policy.

Notice that in all cases, the data is eventually written to stable storage (assuming some failure does not occur to prevent it). The only question is, how much of this process will be completed before the write operation returns and your application can proceed to its next operation.

Setting Durability Guarantees

To set a durability guarantee, use the `Durability` class. When you do this, you must provide three pieces of information:

- The acknowledgment policy.
- A synchronization policy at the master node.
- A synchronization policy at the replica nodes.

The combination of policies that you use is driven by how sensitive your application might be to potential data loss, and by your write performance requirements.

For example, the fastest possible write performance can be achieved through a durability policy that requires:

- No acknowledgments.
- NO_SYNC at the master.
- NO_SYNC at the replicas.

However, this durability policy also leaves your data with the greatest risk of loss due to application or machine failure between the time the operation returns and the time when the data is written to stable storage.

On the other hand, if you want the highest possible durability guarantee, you can use:

- All replicas must acknowledge the write operation.

- SYNC at the master.
- SYNC at the replicas.

Of course, this also results in the slowest possible write performance.

Most commonly, durability policies attempt to strike a balance between write performance and data durability guarantees. For example:

- Simple majority of replicas must acknowledge the write.
- SYNC at the master.
- NO_SYNC at the replicas.

Note that you can set a default durability policy for your `KVStore` handle, but you can also override the policy on a per-operation basis for those situations where some of your data need not be as durable (or needs to be MORE durable) than the default.

For example, suppose you want an intermediate durability policy for most of your data, but sometimes you have transient or easily re-created data whose durability really is not very important. Then you would do something like this:

First, set the default durability policy for the `KVStore` handle:

```
package kvstore.basicExample;

import oracle.kv.Durability;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

Durability defaultDurability =
    new Durability(Durability.SyncPolicy.SYNC,    // Master sync
                   Durability.SyncPolicy.NO_SYNC, // Replica sync
                   Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
kconfig.setDurability(defaultDurability);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

In another part of your code, for some unusual write operations, you might then want to relax the durability guarantee so as to speed up the write performance for those specific write operations:

```
package kvstore.basicExample;

...
```

```
import oracle.kv.Durability;
import oracle.kv.DurabilityException;
import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CLI's 'table create' command.
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Now put all of the cells in the row.

row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Construct a durability policy
Durability durability =
    new Durability(Durability.SyncPolicy.NO_SYNC, // Master sync
                   Durability.SyncPolicy.NO_SYNC, // Replica sync
                   Durability.ReplicaAckPolicy.NONE);

// Construct a WriteOptions object using the durability policy.
WriteOptions wo = new WriteOptions(durability, 0, null);

// Now write the table to the store using the durability policy
// defined, above.
tableH.put(row, null, wo);
```

Chapter 12. Executing a Sequence of Operations

You can execute a sequence of write operations as a single atomic unit so long as all the rows that you are operating upon share the same shard key. By *atomic unit*, we mean all of the operations will execute successfully, or none of them will.

Also, the sequence is performed in isolation. This means that if you have a thread running a particularly long sequence, then another thread cannot intrude on the data in use by the sequence. The second thread will not be able to see any of the modifications made by the long-running sequence until the sequence is complete. The second thread also will not be able to modify any of the data in use by the long-running sequence.

Be aware that sequences only support write operations. You can perform puts and deletes, but you cannot retrieve data when using sequences.

When using a sequence of operations:

- All of the keys in use by the sequence must share the same shard key.
- Operations are placed into a list, but the operations are not necessarily executed in the order that they appear in the list. Instead, they are executed in an internally defined sequence that prevents deadlocks.

The rest of this chapter shows how to use `TableOperationFactory` and `TableAPI.execute()` to create and run a sequence of operations.

Sequence Errors

If any operation within the sequence experiences an error, then the entire operation is aborted. In this case, your data is left in the same state it would have been in if the sequence had never been run at all — no matter how much of the sequence was run before the error occurred.

Fundamentally, there are two reasons why a sequence might abort:

1. An internal operation results in an exception that is considered a fault. For example, the operation throws a `DurabilityException`. Also, if there is an internal failure due to message delivery or a networking error.
2. An individual operation returns normally but is unsuccessful as defined by the particular operation. (For example, you attempt to delete a row that does not exist). If this occurs AND you specified `true` for the `abortIfUnsuccessful` parameter when the operation was created using TableOperationFactory, then an `OperationExecutionException` is thrown. This exception contains information about the failed operation.

Creating a Sequence

You create a sequence by using the `TableOperationFactory` class to create `TableOperation` class instances, each of which represents exactly one operation in the store. You obtain an instance of `TableOperationFactory` by using `TableAPI.getTableOperationFactory()`.

For example, suppose you are using a table defined like this:

```
## Enter into table creation mode
table create -name myTable
## Now add the fields
add-field -type STRING -name itemType
add-field -type STRING -name itemCategory
add-field -type STRING -name itemClass
add-field -type STRING -name itemColor
add-field -type STRING -name itemSize
add-field -type FLOAT -name price
add-field -type INTEGER -name inventoryCount
primary-key -field itemType -field itemCategory -field itemClass
-field itemColor -field itemSize
shard-key -field itemType -field itemCategory -field itemClass
## Exit table creation mode
exit
```

With tables containing data like this:

- Row 1:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: small
  price: 12.07
  inventoryCount: 127

- Row 2:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: medium
  price: 13.07
  inventoryCount: 201

- Row 3:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill
  itemColor: red
  itemSize: large
  price: 14.07
  inventoryCount: 39

And further suppose that this table has rows that require an update (such as a price and inventory refresh), and you want the update to occur in such a fashion as to ensure it is performed consistently for all the rows.

Then you can create a sequence in the following way:

```
package kvstore.basicExample;

import java.util.ArrayList;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import oracle.kv.DurabilityException;
import oracle.kv.FaultException;
import oracle.kv.OperationExecutionException;
import oracle.kv.RequestTimeoutException;

import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableOperationFactory;
import oracle.kv.table.TableOperation;

...

// kvstore handle creation omitted.

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// We use TableOperationFactory to create items for our
// sequence.
TableOperationFactory tof = tableH.getTableOperationFactory();

// This ArrayList is used to contain each item in our sequence.
ArrayList<TableOperation> opList = new ArrayList<TableOperation>();

// Update each row, adding each to the opList as we do.
Row row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "small");
row.put("price", new Float(13.07));
row.put("inventoryCount", 107);
opList.add(tof.createPut(row, null, true));
```

```
row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "medium");
row.put("price", new Float(14.07));
row.put("inventoryCount", 198);
opList.add(tof.createPut(row, null, true));

row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "large");
row.put("price", new Float(15.07));
row.put("inventoryCount", 139);
opList.add(tof.createPut(row, null, true));
```

Note in the above example that we update only those rows that share the same shard key. In this case, the shard key includes the `itemType`, `itemCategory`, and `itemClass` fields. If the value for any of those fields is different from the others, we could not successfully execute the sequence.

Executing a Sequence

To execute the sequence we created in the previous section, use the `TableAPI.execute()` method:

```
package kvstore.basicExample;
try {
    tableH.execute(opList, null);
} catch (OperationExecutionException oee) {
    // Some error occurred that prevented the sequence
    // from executing successfully. Use
    // oee.getFailedOperationIndex() to determine which
    // operation failed. Use oee.getFailedOperationResult()
    // to obtain an OperationResult object, which you can
    // use to troubleshoot the cause of the execution
    // exception.
} catch (IllegalArgumentException iae) {
    // An operation in the list was null or empty.

    // Or at least one operation operates on a row
    // with a shard key that is different
    // than the others.

    // Or more than one operation uses the same key.
} catch (DurabilityException de) {
```

```
    // The durability guarantee could not be met.
} catch (RequestTimeoutException rte) {
    // The operation was not completed inside of the
    // default request timeout limit.
} catch (FaultException fe) {
    // A generic error occurred
}
```

Note that if any of the above exceptions are thrown, then the entire sequence is aborted, and your data will be in the state it would have been in if you had never executed the sequence at all.

`TableAPI.execute()` can optionally take a `WriteOptions` class instance. This class instance allows you to specify:

- The durability guarantee that you want to use for this sequence. If you want to use the default durability guarantee, pass `null` for this parameter.
- A timeout value that identifies the upper bound on the time interval allowed for processing the entire sequence. If you provide `0` to this parameter, the default request timeout value is used.
- A `TimeUnit` that identifies the units used by the timeout value. For example: `TimeUnit.MILLISECONDS`.

For an example of using `WriteOptions`, see Durability Guarantees (page 81).

Appendix A. Third Party Licenses

All of the third party licenses used by Oracle NoSQL Database are described in the `LICENSE.txt` file, which you can find in your KVHOME directory.