Oracle

Getting Started with NoSQL Database

11g Release 2

Library Version 11.2.2.0

Legal Notice

Published 11/26/2012

Table of Contents

Preface

This document describes how to write Oracle NoSQL Database (Oracle NoSQL Database) applications. The APIs used to write an Oracle NoSQL Database application are described here. This book provides the concepts surrounding Oracle NoSQL Database, data schema considerations, as well as introductory programming examples.

This book 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). 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 organized into 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 key-value 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 key-value 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.

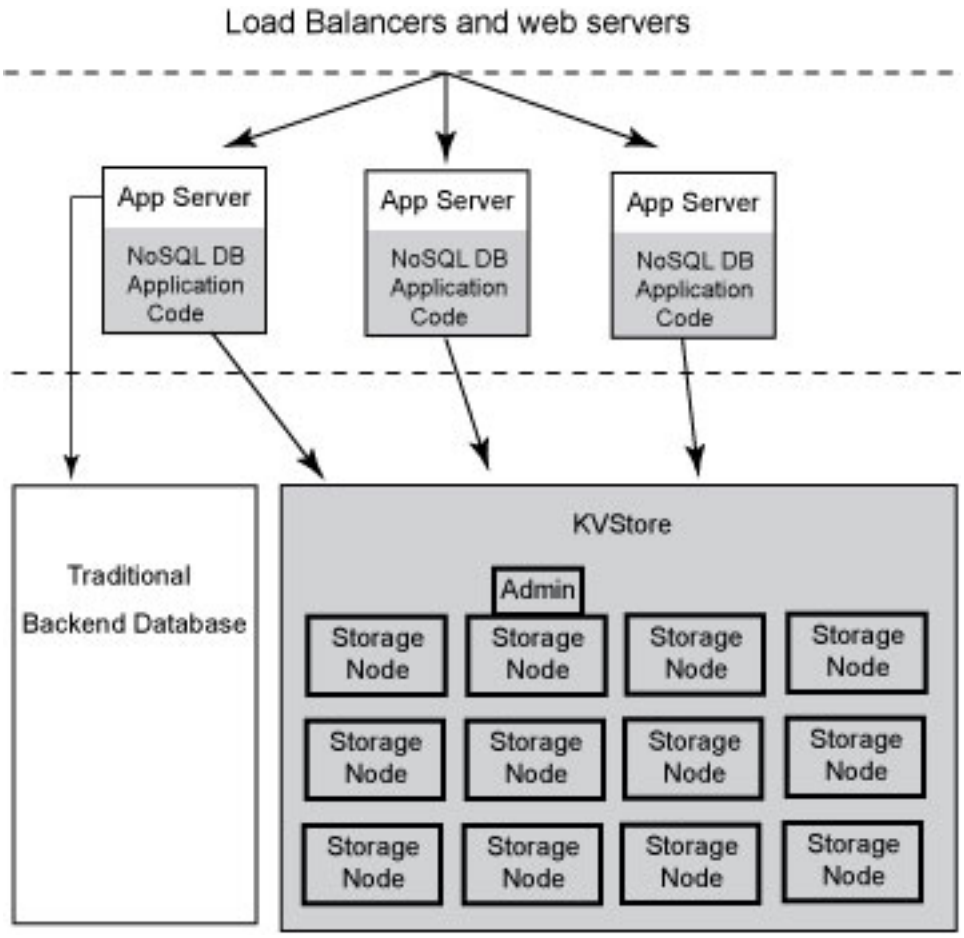
By using the Oracle NoSQL Database APIs, the developer is able to perform create, read, update and delete operations on the data contained in the KVStore. The usage of these APIs is introduced later in this manual.

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:



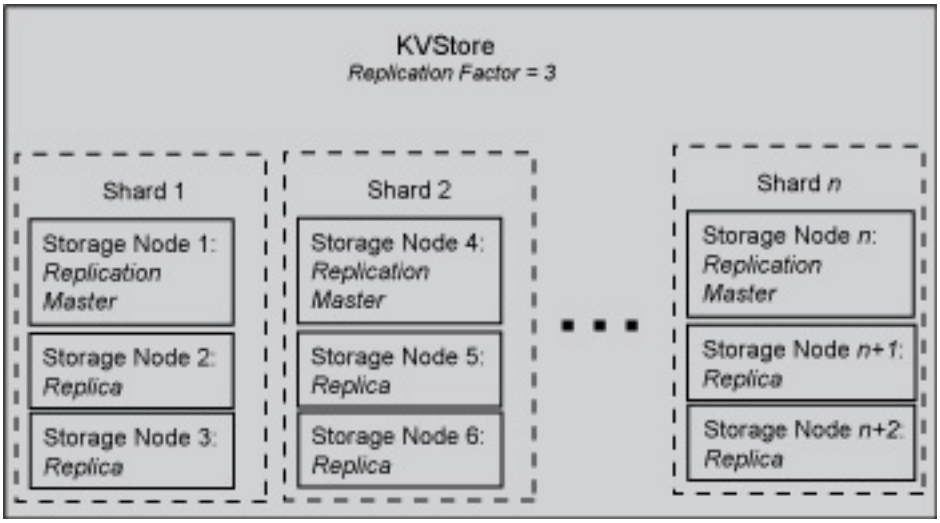
Every Storage Node hosts one or more Replication Nodes, which in turn contain one or more partitions. Also, each Storage Node contains monitoring software that ensures the Replication Nodes which it hosts are running and are otherwise healthy.

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 which is responsible for performing database writes, and which copies those writes to the other Replication Nodes in the shard. This is called the *master* node. All other Replication Nodes in the shard are used to service read-only operations. These are called the *replicas*. Although there can be only one master node at any given time, any of the members of the shard 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.

Note that while no performance tuning of the store is currently possible once it has been deployed, it is also true that how you set your consistency policies and durability guarantees affects the store's overall performance. For more information, see Consistency Guarantees (page 70) and Durability Guarantees (page 77).

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 performance (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). You set the Replication Factor for the store, and then Oracle NoSQL Database makes sure the appropriate number of Replication Nodes are created for each shard that your store contains.

Partitions

Each shard contains one or more *partitions*. Key-value pairs in the store are organized according to the 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 automatically assigns keys evenly across all the available partitions.

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. 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.

Topologies

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

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.

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 malicious access or 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 package as `oracle.kv.util.kvlite.KVLite`, but you can also start it using the `kvlite` utility, found in the `kvstore.jar`. file in the `lib` directory of your Oracle NoSQL Database distribution.

For more information on KVLite, see Introduction to Oracle KVLite (page 6).

Hadoop Integration

Oracle NoSQL Database can be integrated with Apache Hadoop systems using the `oracle.kv.hadoop.KVInputFormat`. This class allows you to read data from Oracle NoSQL Database and then prepare it for insertion into a Hadoop system. To move data in the reverse, you can read data from the Hadoop system using the standard mechanisms, and then write the records to Oracle NoSQL Database using the APIs described in this book.

An example of using `KVInputFormat` to read data from Oracle NoSQL Database in a Map/Reduce job can be found in the `<KVHOME>/examples/hadoop` directory.

In addition, Oracle NoSQL Database provides the `oracle.kv.AvroFormatter` interface. This is used to support Oracle Loader for Hadoop (OLH), which can read data directly from Oracle NoSQL Database and write it to Oracle Database as a Map/Reduce job. OLH is described in the *Oracle Loader for Hadoop* chapter of the *Oracle Big Data Connectors User's Guide*, which you can find here:

http://docs.oracle.com/cd/E27101_01/doc.10/e27365/olh.htm

Chapter 2. Introduction to Oracle KVLite

KVLite is a single-node, single Replication Group 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.

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`.
- Logging is not turned on.
- The administration process is not turned on.

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 -jar KVHOME/lib/kvstore.jar kvlite
```

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 -jar lib/kvclient-M.N.O.jar

  This should write the release to stdout:

  ```
  11gR2.M.N.O...
  ```

- Compile and run the example program:

  1. cd KVHOME
  2. Compile the example:

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

  3. Run the example using all default parameters:

     ```
     java -cp lib/kvclient-M.N.O.jar:examples hello.HelloBigDataWorld
     ```

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

     ```
     java -cp lib/kvclient-M.N.O.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. Record Design Considerations

Oracle NoSQL Database provides the KVStore, which offers storage of key-value pairs. Each such pair can be thought of as a single record in a database, where the key is used to locate the value. Both the key and the value are application-defined, given some loose restrictions imposed by Oracle NoSQL Database.

Every key in the KVStore is a list of strings. All keys must have one or more major components. Keys can also optionally have one or more minor components.

The value portion of the record can be simply a byte array, but in most cases it should use Avro to identify its schema. (See Avro Schemas (page 29) for more information.) The value portion can be as simple or complex as you want it to be.

As a very simple example, suppose you wanted your store to contain information about people. You might then decide to do this:

- Key major: email address.
- Key minor: various properties, such as the user's street address, phone number, photograph, and name.
- Value: Avro-defined information related to the combination of major and minor key components. So, for example, the value for an email address plus a street address might be multiple fields related to street number, street name, city, and so forth.

This is a very simple example of what you might choose to store in Oracle NoSQL Database. However, from a performance point of view, this example might not be the best way for you to organize your data. How you design both your keys and your values can have important performance implications.

The remainder of this chapter describes the performance issues surrounding Oracle NoSQL Database schema design.

Keys

Oracle NoSQL Database organizes records using keys. All records have one or more major key components and, optionally, one or more minor key components. If minor key components are in use, the combination of the major and minor components uniquely identifies a single record in the store.

Keys are spread evenly using a hash across partitions based on the key's major component(s). Every key must have at least one major component, but you can optionally use a list of major components. This means that records that share the same combination of major key components are guaranteed to be in the same partition, which means they can be efficiently queried. In addition, records with identical major key components can be operated upon using multiple operations but under a single atomic operation.

Remember that major key components are used to identify which partition contains a record, and that every partition is stored in a single replication group. This means that major key components are used to identify which replication group stores a given record. The

combination of the major key components, plus the data access operation that you want performed is used to identify which node within the replication group will service the request. Be aware that you cannot control which physical machine, or even which replication group, will be used to store any given piece of data. That is all decided for you by the KV driver.

However, the fact that records are placed on the same physical node based on their major key components means that keys which share major key components can be queried efficiently in a single operation. This is because, conceptually, you are operating on a single physical database when you operate on keys stored together in a single partition. (In reality, a single replication group uses multiple physical databases, but that level of complexity is hidden from you when interacting with the store.)

Remember that every partition is placed in a single replication group, and that your store will probably have multiple replication groups. This is good, because it improves both read and write throughput performance. But in order to take full advantage of that performance enhancement, you need at least as many different major key components as you have partitions. In other words, do not create all your records under a single major key component, or even under a small number of major key components, because doing so will create performance bottle necks as the number of records in your store grow large.

Minor key components also offer performance improvements if used correctly, but in order to understand how you need to understand performance issues surrounding the value portion of your records. We will discuss those issues a little later in this chapter.

What is a Key Component?

A key component is a Java String. Issues of comparison can be answered by examining how Java Strings are compared using your preferred encoding.

Because it is a String, a key component can be anything you want it to be. Typically, some naming scheme is adopted for the application so as to logically organize records.

It helps to think of key components as being locations in a file system path. You can write out a record's components as if they were a file system path delimited by a forward slash ("/"). For example, suppose you used multiple major components to identify a record, and one such record using the following major components: "Smith", and "Bob." Another record might use "Smith" and "Patricia". And a third might use "Wong", and "Bill". Then the major components for those records could be written as:

```
/Smith/Bob
/Smith/Patricia
/Wong/Bill
```

Further, suppose you had different kinds of information about each user that you want to store. Then the different types of information could be further identified using minor components such as "birthdate", "image", "phonenumber", "userID", and so forth. The minor portion of a key component is separated by the major components by a special slash-hyphen-slash delimiter (`/-/`).

By separating keys into major and minor key components, we could potentially store and operate upon the following records. Those that share a common major component can be operated upon in a single atomic operation:

```
/Smith/Bob/-/birthdate
/Smith/Bob/-/phonenumber
/Smith/Bob/-/image
/Smith/Bob/-/userID
/Smith/Patricia/-/birthdate
/Smith/Patricia/-/phonenumber
/Smith/Patricia/-/image
/Smith/Patricia/-/userID
/Wong/Bill/-/birthdate
/Wong/Bill/-/phonenumber
/Wong/Bill/-/image
/Wong/Bill/-/userID
```

Note that the above keys might not represent the most efficient way to organize your data. We discuss this issue in the next section.

Values

Records in the store are organized as key-value pairs. The *value* is the data that you want to store, manage and retrieve. In almost all cases, you should create your values using Avro to describe the value's data schema. Avro schemas are described in Avro Schemas (page 29).

In simple cases, values can also be organized as a byte array. If so, the mapping of the arrays to data structures (serialization and deserialization) is left entirely to the application. Note that while the earliest portions of this manual show byte array usage in its examples, this is not the recommended approach. Instead, you should use Avro even for very simple values.

There are no restrictions on the size of your values. However, you should consider your store's performance when deciding how large you are willing to allow your individual records to become. As is the case with any data storage scheme, the larger your record, the longer it takes to read the information from storage, and to write the information to storage. If your values become so large that they impact store read/write performance, or are even too large to fit into your memory cache (or even your Java heap space) then you should consider storing your values using Oracle NoSQL Database's large object support. See Using Large Objects (page 88) for details.

On the other hand, every record carries with it some amount of overhead. Also, as the number of your records grows very large, search times may begin to be adversely affected. As a result, choosing to store an extremely large number of very small records can also harm your store's performance.

Therefore, when designing your store's content, you must find the appropriate balance between a small number of very large records and a large number of very small records. You should also consider how frequently any given piece of information will be accessed.

For example, suppose your store contains information about users, where each user is identified by their email address. 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 record. (This is unlikely, but it does represent the simplest case.) In that event, you might create a single Avro schema that represents each of the properties you maintain for the users in your application. You can then trivially organize your records using only major key components so that, for example, all of the data for user Bob Smith can be accessed using the key `/Smith/Bob`.

However, the chances are good that your application will not require you to access *all* of the properties for a user's record every time you access that record. 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 use a key other than that used by the frequently accessed properties. The different keys for these large properties can share major key components, while differing in their minor key components. However, if you are using large object support for your large properties, then these must be under a major key that is different from the major key you use for the other properties you are storing.

At the same time, there is overhead involved with every key your store contains, so you do not want to create a key for every possible user property. For this reason, if you have a lot of small properties, you might want to organize them all under a single key even if only some of them are likely to be updated or read for any given operation.

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 record 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 four keys per user. Each key shares the same major components, and differs in its minor component, in the following way:

1. /*surname*/*familiar name*/-/contact

   The value for this key is a Avro record that contains all of the small user properties (name, phone number, address, and so forth).

2. /*surname*/*familiar name*/-/publickeys

   The value for this key is an Avro record that contains the user's public keys. These are always read and written at the same time, so it makes sense to organize them under one key.

3. /image.lob/-/*surname*/*familiar name*

   The value for this key is an image file, saved using Oracle NoSQL Database's large object support.

4. /audio.lob/-/voicegreeting/*surname*/*familiar name*

   The value for this key is an mp3 file, also saved using the large object support.

Any data organized under different keys which differ only in the minor key component allows you to read and update the various properties all at once using a single atomic operation, which gives you full ACID support for user record updates. At the same time, your application does not have to be reading and writing large properties (image files, voice recordings, and so forth) unless it is absolutely necessary. When it is necessary to read or write these large objects, you can use the Oracle NoSQL Database stream interface which is optimized for that kind of traffic.

Chapter 4. Developing for Oracle NoSQL Database

You access the data in the Oracle NoSQL Database KVStore using Java APIs that are provided with the product. These APIs allow you to create keys and values, put key-value pairs into the store, then retrieve them again. You also use these APIs to define consistency and durability guarantees. It is also possible to execute a sequence of store operations at one time 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.

Note

Oracle NoSQL Database requires Java SE 6 (jdk 1.6.0 u25) or later.

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` class.

When you get a `KVStore` handle from the `KVStoreFactory` class, 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 replication nodes currently belonging to the replication group. 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.

```
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 70).

- 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 shutdown abnormally. For more information, see Durability Guarantees (page 77).

- 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. 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.

Chapter 5. Writing and Deleting Records

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

Write Exceptions

There are three 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 77).

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 `KVStore.put()` method that allows you to specify a timeout value for that specific operation.)

Finally, you can receive a `FaultException`, which indicates that some exception occurred that cannot be handled by your application. Your only recourse here is to either log the error and move along, or retry the operation.

Writing Records to the Store

Creating a new record in the store and updating an existing record are usually identical operations (although methods exist that work only if the record is being updated, or only if it is being created — these are described a little later in this section). In both cases, you simply write a record to the store that uses the appropriate key. If a record with that key does not currently exist in the store, then the record is created for you. If a record exists that does use the specified key, then that record is updated with the information that you are writing to the store.

In order to put an ordinary record into the store:

1. Construct a key, making sure to specify all of the key's major and minor path components. For information on major and minor path components, see Record Design Considerations (page 9).
2. Construct a value. This is the actual data that you want to put into the store. Normally you want to use the Avro data format for this, which is described in Avro Schemas (page 29) and Avro Bindings (page 44).

3. Use one of the `KVStore` class's put methods to put the record to the store.

The following is a trivial example of writing a record to the store. It assumes that the `KVStore` handle has already been created. For the sake of simplicity, this example trivially serializes a string to use as the value for the put operation. This is *not* what you should do in your code. Rather, you should use the Avro data format, even for this trivial case.

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import oracle.kv.Value;
import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

minorComponents.add("phonenumber");

// Create the key
Key myKey = Key.createKey(majorComponents, minorComponents);

String data = "408 555 5555";

// Create the value. Notice that we serialize the contents of the
// String object when we create the value.
Value myValue = Value.createValue(data.getBytes());

// Now put the record. Note that we do not show the creation of the
// kvstore handle here.

kvstore.put(myKey, myValue);
```

Other put Operations

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

- `KVStore.putIfAbsent()`

  This method will only put the record if the key DOES NOT current exist in the store. That is, this method is successful only if it results in a *create* operation.

- `KVStore.putIfPresent()`

  This method will only put the record if the key already exists in the store. That is, this method is only successful if it results in an *update* operation.

- `KVStore.putIfVersion()`

  This method will put the record only if the value matches the supplied version information. For more information, see Using Versions (page 68).

Deleting Records from the Store

You delete a single record from the store using the `KVStore.delete()` method. Records are deleted based on a key. You can also require a record to match a specified version before it will be deleted. To do this, use the `KVStore.deleteIfVersion()` method. Versions are described in Using Versions (page 68).

When you delete a record, you must handle the same exceptions as occur when you perform any write operation on the store. See Write Exceptions (page 16) for a high-level description of these exceptions.

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

minorComponents.add("phonenumber");

// Create the key
Key myKey = Key.createKey(majorComponents, minorComponents);

// Now delete the record. Note that we do not show the creation of the
// kvstore handle here.

kvstore.delete(myKey);
```

Using multiDelete()

You can delete multiple records at once, so long as they all share the same major path components. Note that you must provide a complete major path component. You can omit minor path components, or even provide partial path components.

To delete multiple records at once, use the `KVStore.multiDelete()` method.

For example:

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

// Create the key
Key myKey = Key.createKey(majorComponents);

// Now delete the record. Note that we do not show the creation of the
// kvstore handle here.

kvstore.multiDelete(myKey, null, null);
```

Chapter 6. Reading Records

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

1. Retrieve a single record at a time using `KVStore.get()`.
2. Retrieve records that share a complete set of major components using either `KVStore.multiGet()` or `KVStore.multiGetIterator()`.
3. Retrieve records that share a partial set of major components using `KVStore.storeIterator()`.

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 70).

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 replication groups in your store, with the partitioning occurring based on the major path components in your 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 replication groups 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. (There is a version of the `KVStore.get()` method that allows you to specify a timeout value for that specific operation.)

Finally, you can receive a `FaultException`, which indicates that some exception occurred that cannot be handled by your application. Your only recourse here is to either log the error and move along, or retry the operation.

Retrieving a Single Record

To retrieve a record from the store, use the `KVStore.get()` method. This method returns a `ValueVersion` object. Use `ValueVersion.getValue()` to return the `Value` object associated with the key. It is then up to your application to turn the `Value`'s byte array into a useful form. Normally, this will require the use of an Avro binding. See Avro Bindings (page 44) for details.

For example, in Writing Records to the Store (page 16) we showed a trivial example of storing a key-value pair to the store, where the value was a simple String. The following trivial example shows how to retrieve that record. (Again, this is *not* how your code should deserialize data, because this example does not use Avro to manage the value's schema.)

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.ValueVersion;
import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

minorComponents.add("phonenumber");

// Create the key
Key myKey = Key.createKey(majorComponents, minorComponents);

// Now retrieve the record. Note that we do not show the creation of
// the kvstore handle here.

ValueVersion vv = kvstore.get(myKey);
Value v = vv.getValue();
String data = new String(v.getValue());
```

Using multiGet()

`KVStore.multiGet()` allows you to retrieve multiple records at once, so long as they all share the same major path components. The major path components that you provide must represent a *complete* set of components.

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

For example, suppose you use the following keys:

```
/Hats/-/baseball
/Hats/-/baseball/longbill
/Hats/-/baseball/longbill/blue
```

```
/Hats/-/baseball/longbill/red
/Hats/-/baseball/shortbill
/Hats/-/baseball/shortbill/blue
/Hats/-/baseball/shortbill/red
/Hats/-/western
/Hats/-/western/felt
/Hats/-/western/felt/black
/Hats/-/western/felt/gray
/Hat/-/swestern/leather
/Hat/-/swestern/leather/black
/Hat/-/swestern/leather/gray
```

Then you can retrieve all of the records that use the major key component `Hats` as follows:

```
package kvstore.basicExample;

...

import oracle.kv.ConsistencyException;
import oracle.kv.Key;
import oracle.kv.RequestTimeoutException;
import oracle.kv.Value;
import oracle.kv.ValueVersion;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.SortedMap;
import java.util.Map;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Hats");

// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.

SortedMap<Key, ValueVersion> myRecords = null;

try {
    myRecords = kvstore.multiGet(myKey, 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 sorted map as follows:

```
for (Map.Entry<Key, ValueVersion> entry : myRecords.entrySet()) {
    ValueVersion vv = entry.getValue();
    Value v = vv.getValue();
    // Do some work with the Value here
}
```

Using multiGetIterator()

If you believe your return set will be so large that it cannot fit into memory, use `KVStore.multiGetIterator()` instead of `KVStore.multiGet()`.

`KVStore.multiGetIterator()` allows you to perform an ordered traversal of a set of keys, as defined by a key and, optionally, a key range. Use this method if you believe your return set will not fit into memory, or if you believe the return set will be so large that it might strain your network resources.

`KVStore.multiGetIterator()` does not return the entire set of records all at once. Instead, it batches the fetching of key-value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

Note that this method does not result in a transactional 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 ordered traversal of records that share the same major path components. The major path components that you provide must represent a *complete* set of components.

To use this method, you must provide:

- A traversal direction.
- 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.
- The key whose child pairs are to be fetched.

Note that there are other possible parameters that you can provide, but this above list represents the minimum information required to use this method.

For example, suppose the following is representative of the keys that you use:

```
/Hats/-/baseball
/Hats/-/baseball/longbill
```

```
/Hats/-/baseball/longbill/blue
/Hats/-/baseball/longbill/red
/Hats/-/baseball/shortbill
/Hats/-/baseball/shortbill/blue
/Hats/-/baseball/shortbill/red
/Hats/-/western
/Hats/-/western/felt
/Hats/-/western/felt/black
/Hats/-/western/felt/gray
/Hats/-/western/leather
/Hats/-/western/leather/black
/Hats/-/western/leather/gray
```

Then you can retrieve all of the records that use the major key component `Hats` as follows:

```
package kvstore.basicExample;

...

import oracle.kv.Direction;
import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.KeyValueVersion;

import java.util.ArrayList;
import java.util.Iterator;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Hats");

// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.


Iterator<KeyValueVersion> i =
    kvstore.multiGetIterator(Direction.FORWARD, 0,
                             myKey, null, null);
while (i.hasNext()) {
    Value v = i.next().getValue();
    // Do some work with the Value here
}
```

Using storeIterator()

If you want to retrieve all the records that match only some of the major key components, use `KVStore.storeIterator()`. Using this method, you can iterate over all of the records in the store, or over all of the records that match a partial set of major components.

`KVStore.storeIterator()` does not return the entire set of records all at once. Instead, it batches the fetching of key-value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth. Also, the records 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 records in the store. If you do not provide a key, then this method will iterate over all of the records in the store. If you do provide a key, you must provide only a subset of the major key components used by your records. The key that you provide must NOT include any minor key components.

To use this method, you must specify:

- A traversal direction.
- 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.

For example, suppose you are storing user records that use keys like this:

```
/Smith/Bob/-/birthdate
/Smith/Bob/-/phonenumber
/Smith/Bob/-/image
/Smith/Bob/-/userID
/Smith/Patricia/-/birthdate
/Smith/Patricia/-/phonenumber
/Smith/Patricia/-/image
/Smith/Patricia/-/userID
/Smith/Richard/-/birthdate
/Smith/Richard/-/phonenumber
/Smith/Richard/-/image
/Smith/Richard/-/userID
/Wong/Bill/-/birthdate
/Wong/Bill/-/phonenumber
/Wong/Bill/-/image
/Wong/Bill/-/userID
```

Then you can retrieve all of the records for all users whose surname is 'Smith' as follows:

```
package kvstore.basicExample;

...
```

```
import oracle.kv.Direction;
import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.KeyValueVersion;
import java.util.ArrayList;
import java.util.Iterator;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");

// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.


Iterator <KeyValueVersion>i =
    kvstore.storeIterator(Direction.UNORDERED, 0,
                          myKey, null, null);
while (i.hasNext()) {
    Value v = i.next().getValue();
    // Do some work with the Value here
}
```

Specifying Subranges

When performing multi-key operations in the store, you can specify a range of records to operate upon. You do this using the `KeyRange` class. This class defines a range of `String` values for the key components immediately following a key that is used in a multiple get operation.

For example, suppose you were using the following keys:

```
/Smith/Bob/-/birthdate
/Smith/Bobphone/-/number
/Smith/Bob/-/image
/Smith/Bob/-/userID
/Smith/Patricia/-/birthdate
/Smith/Patricia/-/phonenumber
/Smith/Patricia/-/image
/Smith/Patricia/-/userID
/Smith/Richard/-/birthdate
```

```
/Smith/Richard/-/phonenumber
/Smith/Richard/-/image
/Smith/Richard/-/userID
/Wong/Bill/-/birthdate
/Wong/Bill/-/phonenumber
/Wong/Bill/-/image
/Wong/Bill/-/userID
```

Given this, you could perform operations for all the records related to users Bob Smith and Patricia Smith by constructing a `KeyRange`. When you do this, you must identify the key components that defines the upper and lower bounds of the range. You must also identify if the key components that you provide are inclusive or exclusive.

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.

```
package kvstore.basicExample;

...

import oracle.kv.KeyRange;

...

KeyRange kr = new KeyRange("Bob", true, "Patricia", true);
```

You then use the `KeyRange` instance when you perform your multi-key operation. For example, suppose you you were retrieving records from your store using `KVStore.storeIterator()`:

```
package kvstore.basicExample;

...

import oracle.kv.Direction;
import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.KeyRange;
import oracle.kv.KeyValueVersion;

import java.util.ArrayList;
import java.util.Iterator;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
```

```
// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

KeyRange kr = new KeyRange("Bob", true, "Patricia", true);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.


Iterator<KeyValueVersion> i =
    kvstore.storeIterator(Direction.UNORDERED, 0,
                          myKey, kr, null);
while (i.hasNext()) {
    Value v = i.next().getValue();
    // Do some work with the Value here
}
```

Chapter 7. Avro Schemas

Avro is used to define the data schema for a record's value. This schema describes the fields allowed in the value, along with their data types.

You apply a schema to the value portion of an Oracle NoSQL Database record using Avro bindings. These bindings are used to serialize values before writing them, and to deserialize values after reading them. The usage of these bindings requires your applications to use the Avro data format, which means that each stored value is associated with a schema.

The use of Avro schemas allows serialized values to be stored in a very space-efficient binary format. Each value is stored without any metadata other than a small internal schema identifier, between 1 and 4 bytes in size. One such reference is stored per key-value pair. In this way, the serialized Avro data format is always associated with the schema used to serialize it, with minimal overhead. This association is made transparently to the application, and the internal schema identifier is managed by the bindings supplied by the AvroCatalog class. The application never sees or uses the internal identifier directly.

The Avro API is the result of an open source project provided by the Apache Software Foundation. It is formally described here: http://avro.apache.org.

In addition, Avro makes use of the Jackson APIs for parsing JSON. This is likely to be of interest to you if you are integrating Oracle NoSQL Database with a JSON-based system. Jackson is formally described here: http://wiki.fasterxml.com/JacksonHome.

Creating Avro Schemas

An Avro schema is created using JSON format. JSON is short for *JavaScript Object Notation*, and it is a lightweight, text-based data interchange format that is intended to be easy for humans to read and write. JSON is described in a great many places, both on the web and in after-market documentation. However, it is formally described in the IETF's RFC 4627, which can be found at http://www.ietf.org/rfc/rfc4627.txt?number=4627.

To describe an Avro schema, you create a JSON record which identifies the schema, like this:

```
{
     "type": "record",
     "namespace": "com.example",
     "name": "FullName",
     "fields": [
       { "name": "first", "type": "string" },
       { "name": "last", "type": "string" }
     ]
}
```

The above example is a JSON record which describes schema that might be used by the value portion of a key-value pair in the store. It describes a schema for a person's full name.

Notice that for the record, there are four fields:

- `type`

  Identifies the JSON field type. For Avro schemas, this must always be `record` when it is specified at the schema's top level. The type `record` means that there will be multiple fields defined.

- `namespace`

  This identifies the namespace in which the object lives. Essentially, this is meant to be a URI that has meaning to you and your organization. It is used to differentiate one schema type from another should they share the same name.

- `name`

  This is the schema name which, when combined with the namespace, uniquely identifies the schema within the store. In the above example, the fully qualified name for the schema is `com.example.FullName`.

- `fields`

  This is the actual schema definition. It defines what fields are contained in the value, and the data type for each field. A field can be a simple data type, such as an integer or a string, or it can be complex data. We describe this in more detail, below.

  Note that schema field names must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_].

To use the schema, you must define it in a flat text file, and then add the schema to your store using the appropriate command line call. You must also somehow provide it to your code. The schema that your code is using must correspond to the schema that has been added to your store.

The remainder of this chapter describes schemas and how to add them to your store. For a description of how to use schemas in your code, see Avro Bindings (page 44).

Avro Schema Definitions

Avro schema definitions are JSON records. Because it is a record, it can define multiple fields which are organized in a JSON array. Each such field identifies the field's name as well as its type. The type can be something simple, like an integer, or something complex, like another record.

For example, the following trivial Avro schema definition can be used for a value that contains just someone's age:

```
{
    "type" : "record",
    "name" : "userAge",
    "namespace" : "my.example",
    "fields" : [{"name" : "age", "type" : "int"}]
}
```

Of course, if your data storage needs are this simple, you can just use a byte-array to store the integer in the store. (Although this is not considered best practice.)

Notice in the previous example that the top-level type for the schema definition is of type `record`, even though we are defining a single-field schema. Oracle NoSQL Database requires you to use `record` for the top-level type, even if you only need one field.

Also, it is best-practice to define default values for the fields in your schema. While this is optional, should you ever decide to change your schema, it can save you a lot of trouble. To define a default value, use the `default` attribute:

```
{
    "type" : "record",
    "name" : "userAge",
    "namespace" : "my.example",
    "fields" : [{"name" : "age", "type" : "int", "default" : -1}]
}
```

You almost certainly will not be using single-field definitions. To add multiple fields, specify an array in the `fields` field. For example:

```
{
    "type" : "record",
    "name" : "userInformation",
    "namespace" : "my.example",
    "fields" : [{"name" : "username",
                 "type" : "string",
                 "default" : "NONE"},

                 {"name" : "age",
                 "type" : "int",
                 "default" : -1},

                 {"name" : "phone",
                 "type" : "string",
                 "default" : "NONE"},

                 {"name" : "housenum",
                 "type" : "string",
                 "default" : "NONE"},

                 {"name" : "street",
                 "type" : "string",
                 "default" : "NONE"},

                 {"name" : "city",
                 "type" : "string",
                 "default" : "NONE"},

                 {"name" : "state_province",
                 "type" : "string",
```

```
                "default" : "NONE"},

                {"name" : "country",
                "type" : "string",
                "default" : "NONE"},

                {"name" : "zip",
                "type" : "string",
                "default" : "NONE"}]
}
```

The above schema definition provides a lot of information. However, simple as it is, you could add some more structure to it by using an embedded record:

```
{
    "type" : "record",
    "name" : "userInformation",
    "namespace" : "my.example",
    "fields" : [{"name" : "username",
                 "type" : "string",
                 "default" : "NONE"},

                {"name" : "age",
                 "type" : "int",
                 "default" : -1},

                 {"name" : "phone",
                  "type" : "string",
                  "default" : "NONE"},

                 {"name" : "housenum",
                  "type" : "string",
                  "default" : "NONE"},

                  {"name" : "address",
                     "type" : {
                         "type" : "record",
                         "name" : "mailing_address",
                         "fields" : [
                            {"name" : "street",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "city",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "state_prov",
                             "type" : "string",
                             "default" : "NONE"},
```

```
                            {"name" : "country",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "zip",
                             "type" : "string",
                             "default" : "NONE"}
                            ]}
                }
    ]
}
```

Note

It is unlikely that you will need just one record definition for your entire store. Probably you will have more than one type of record. You handle this by providing each of your record definitions individually in separate files. Your code must then be written to handle the different record definitions. We will discuss how to do that later in this chapter.

Primitive Data Types

In the previous Avro schema examples, we have only shown strings and integers. The complete list of primitive types which Avro supports are:

- `null`

  No value.

- `boolean`

  A binary value.

- `int`

  A 32-bit signed integer.

- `long`

  A 64-bit signed integer.

- `float`

  A single precision (32 bit) IEEE 754 floating-point number.

- `double`

  A double precision (64-bit) IEEE 754 floating-point number.

- `bytes`

A sequence of 8-bit unsigned bytes.

- `string`

  A unicode character sequence.

These primitive types do not have any specified attributes. Primitive type names are also defined type names. For example, the schema "string" is equivalent to:

```
{"type" : "string"}
```

Complex Data Types

Beyond the primitive data types described in the previous section, Avro also supports six complex data types: Records, Enums, Arrays, Maps, Unions, and Fixed. They are described in this section.

record

A record represents an encapsulation of attributes that, all combined, describe a single thing. The attributes that an Avro record supports are:

- name

  This is the record's name, and it is required. It is meant to identify the thing that the record describes. For example: `PersonInformation` or `Automobiles` or `Hats` or `BankDeposit`.

  Note that record names must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_].

- namespace

  A namespace is an optional attribute that uniquely identifies the record. It is optional, but it should be used when there is a chance that the record's name will collide with another record's name. For example, suppose you have a record that describes an employee. However, you might have several different types of employees: fulltime, part time, and contractors. So you might then create all three types of records with the name `EmployeeInfo`, but then with namespaces such as `FullTime`, `PartTime` and `Contractor`. The fully qualified name for the records used to describe full time employees would then be `FullTime.EmployeeInfo`.

  Alternatively, if your store contains information for many different organizations, you might want to use a namespace that identifies the organization used by the record so as to avoid collisions in the record names. In this case, you could end up with fully qualified records with names such as `My.Company.Manufacturing.EmployeeInfo` and `My.Company.Sales.EmployeeInfo`.

- doc

  This optional attribute simply provides documentation about the record. It is parsed and stored with the schema, and is available from the Schema object using the Avro API, but it is not used during serialization.

- aliases

  This optional attribute provides a JSON array of strings that are alternative names for the record. Note that there is no such thing as a rename operation for JSON schema. So if you want to refer to a schema by a name other than what you initially defined in the `name` attribute, use an alias.

- type

  A required attribute that is either the keyword `record`, or an embedded JSON schema definition. If this attribute is for the top-level schema definition, `record` must be used.

- fields

  A required attribute that provides a JSON array which lists all of the fields in the schema. Each field must provide a name and a type attribute. Each field may provide doc, order, aliases and default attributes:

  - The name, type, doc and aliases attributes are used in the exact same way as described earlier in this section.

    As is the case with record names, field names must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_].

  - The order attribute is optional, and it is ignored by Oracle NoSQL Database. For applications (other than Oracle NoSQL Database) that honor it, this attribute describes how this field impacts sort ordering of this record. Valid values are `ascending`, `descending`, or `ignore`. For more information on how this works, see http://http://avro.apache.org/docs/current/spec.html#order.

  - The default attribute is optional, but highly recommended in order to support schema evolution. It provides a default value for the field that is used only for the purposes of schema evolution. Use of the default attribute does not mean that you can fail to initialize the field when creating a new value object; all fields must be initialized regardless of whether the default attribute is present.

    Schema evolution is described in Schema Evolution (page 37).

    Permitted values for the default attribute depend on the field's type. Default values for unions depend on the first field in the union. Default values for bytes and fixed fields are JSON strings.

Enum

Enums are enumerated types, and it supports the following attributes

- name

  A required attribute that provides the name for the enum. This name must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_].

- namespace

An optional attribute that qualifies the enum's name attribute.

- aliases

  An optional attribute that provides a JSON array of alternative names for the enum.

- doc

  An optional attribute that provides a comment string for the enum.

- symbols

  A required attribute that provides the enum's symbols as an array of names. These symbols must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_].

For example:

```
{ "type" : "enum",
  "name" : "Colors",
  "namespace" : "palette",
  "doc" : "Colors supported by the palette.",
  "symbols" : ["WHITE", "BLUE", "GREEN", "RED", "BLACK"]}
```

Arrays

Defines an array field. It only supports the items attribute, which is required. The items attribute identifies the type of the items in the array:

```
{"type" : "array", "items" : "string"}
```

Maps

A map is an associative array, or dictionary, that organizes data as key-value pairs. The key for an Avro map must be a string. Avro maps supports only one attribute: values. This attribute is required and it defines the type for the value portion of the map.

```
{"type" : "map", "values" : "int"}
```

Unions

A union is used to indicate that a field may have more than one type. They are represented as JSON arrays.

For example, suppose you had a field that could be either a string or null. Then the union is represented as:

```
["string", "null"]
```

You might use this in the following way:

```
{
     "type": "record",
     "namespace": "com.example",
     "name": "FullName",
```

```
      "fields": [
        { "name": "first", "type": ["string", "null"] },
        { "name": "last", "type": "string", "default" : "Doe" }
      ]
}
```

Fixed

A fixed type is used to declare a fixed-sized field that can be used for storing binary data. It has two required attributes: the field's name, and the size in 1-byte quantities.

For example, to define a fixed field that is one kilobyte in size:

```
{"type" : "fixed" , "name" : "bdata", "size" : 1048576}
```

Using Avro Schemas

Once you have defined your schema, you make use of it in your Oracle NoSQL Database application in the following way:

1. Add the schema to your store. See Managing Avro Schema in the Store (page 41) for information on how to do this.
2. Identify the schema to your application.
3. Serialize and/or deserialize Oracle NoSQL Database values which use the Avro data format. You use Avro bindings to perform the serialization functions. There are different bindings available to you, each of which offers pluses and negatives. We will describe the different bindings later in this section.

Other than that, the mechanisms you use to read/write/delete records in the store do not change just because you are using the Avro data format with your values. Avro affects your code only where you manage your values.

The following sections describe the bindings that you use to serialize and deserialize your data. The binding that you use defines how you provide your schema to the store.

Schema Evolution

Schema evolution is the term used for how the store behaves when Avro schema is changed after data has been written to the store using an older version of that schema. To change an existing schema, you update the schema as stored in its flat-text file, then add the new schema to the store using the `ddl add-schema` command with the `-evolve` flag.

For example, if a middle name property is added to the FullName schema, it might be stored in a file named `schema2.avsc`, and then added to the store using the `ddl add-schema` command.

Note that when you change schema, the new field must be given a default value. This prevents errors when clients using an old version of the schema create new values that will be missing the new field:

```
{
   "type": "record",
   "namespace": "com.example",
   "name": "FullName",
   "fields": [
     { "name": "first", "type": "string" },
     { "name": "middle", "type": "string", "default": "" },
     { "name": "last", "type": "string" }
   ]
 }
```

These are the modifications you can safely perform to your schema without any concerns:

- A field with a default value is added.
- A field that was previously defined with a default value is removed.
- A field's doc attribute is changed, added or removed.
- A field's order attribute is changed, added or removed.
- A field's default value is added, or changed.
- Field or type aliases are added, or removed.
- A non-union type may be changed to a union that contains only the original type, or vice-versa.

Beyond these kind of changes, there are unsafe changes that you can do which will either cause the schema to be rejected when you attempt to add it to the store, or which can be performed so long as you are careful about how you go about upgrading clients which use the schema. These type of issues are identified when you try to modify (evolve) schema that is currently enabled in the store. See Changing Schema (page 42) for details.

Rules for Changing Schema

There are a few rules you need to remember if you are modifying schema that is already in use in your store:

1. For best results, always provide a default value for the fields in your schema. This makes it possible to delete fields later on if you decide it is necessary. *If you do not provide a default value for a field, you cannot delete that field from your schema.*
2. You cannot change a field's data type. If you have decided that a field should be some data type other than what it was originally created using, then add a whole new field to your schema that uses the appropriate data type.
3. When adding a field to your schema, you must provide a default value for the field.
4. You cannot rename an existing field. However, if you want to access the field by some name other than what it was originally created using, add and use aliases for the field.

Writer and Reader Schema

When a schema is changed, multiple versions of the schema will exist and be maintained by the store. The version of the schema used to serialize a value, before writing it to the store, is called the *writer schema*. The writer schema is specified by the application when creating a binding. It is associated with the value when calling the binding's `AvroBinding.toValue()` method to serialize the data. This writer schema is associated internally with every stored value.

The *reader schema* is used to deserialize a value after reading it from the store. Like the writer schema, the reader schema is specified by the client application when creating a binding. It is used to deserialize the data when calling the binding's `AvroBinding.toObject()` method, after reading a value from the store.

How Schema Evolution Works

Schema evolution is the automatic transformation of Avro schema. This transformation is between the version of the schema that the client is using (its local copy), and what is currently contained in the store. When the local copy of the schema is not identical to the schema used to write the value (that is, when the reader schema is different from the writer schema), this data transformation is performed. When the reader schema matches the schema used to write the value, no transformation is necessary.

Schema evolution is applied only during deserialization. If the reader schema is different from the value's writer schema, then the value is automatically modified during deserialization to conform to the reader schema. To do this, default values are used.

There are two cases to consider when using schema evolution: when you add a field and when you delete a field. Schema evolution takes care of both scenarios, so long as you originally assigned default values to the fields that were deleted, and assigned default values to the fields that were added.

Adding Fields

Suppose you had the following schema:

```
{
    "type" : "record",
    "name" : "userInfo",
    "namespace" : "my.example",
    "fields" : [{"name" : "name", "type" : "string", "default" : ""}]
}
```

In version 2 of the schema, you add a field:

```
{
    "type" : "record",
    "name" : "userAge",
    "namespace" : "my.example",
    "fields" : [{"name" : "name", "type" : "string", "default" : ""},
                {"name" : "age", "type" : "int" , "default" : -1}]
```

```
}
```

In this scenario, a client that is using the new schema can deserialize a value that uses the old schema, even though the age field will be missing from the value. Upon deserialization, the value retrieved from the store will be automatically transformed such that the age field is contained in the value. The age field will be set to the default value, which is -1 in this case.

The reverse also works. A client that is using the old version of the schema attempts can deserialize a value that was written using the new version of the schema. In this case, the value retrieved from the store contains the age field, which from the client perspective is unexpected. So upon deserialization the age field is automatically removed from the retrieved object.

This has ramifications if you change your schema, and then have clients concurrently running that are using different schema versions. This scenario is not unusual in a large, distributed system of the type that Oracle NoSQL Database supports.

In this scenario, you might see fields revert to their default value, even though no client has explicitly touched those fields. This can happen in the following way:

1. Client v.2 creates a my.example.userInfo record, and sets the age field to 38. Then it writes that value to the store. Client v.2 is using schema version 2.

2. Client v.1 reads the record. It is using version 1 of the schema, so the age field is automatically removed from the value during deserialization.

   Client v.1 modifies the name field and then writes the record back to the store. When it does this, the age field is missing from the value that it writes to the store.

3. Client v.2 reads the record again. Because the age field is missing from the record (because Client v.1 last wrote it), the age field is set to the default value, which is -1. This means that the value of the age field has reverted to the default, even though no client explicitly modified it.

Deleting Fields

Field deletion works largely the same way as field addition, with the same concern for field values automatically reverting to the default. Suppose you had the following trivial schema:

```
{
    "type" : "record",
    "name" : "userAge",
    "namespace" : "my.example",
    "fields" : [{"name" : "name", "type" : "string", "default" : ""},
                {"name" : "age", "type" : "int" , "default" : -1}]

}
```

In version 2 of the schema, you delete the age field:

```
{
    "type" : "record",
```

```
    "name" : "userInfo",
    "namespace" : "my.example",
    "fields" : [{"name" : "name", "type" : "string", "default" : ""}]
}
```

In this scenario, a client that is using the new schema can deserialize a value that uses the old schema, even though the age field is contained in that value. In this case, the age field is silently removed from the value during deserialization.

Further, a client that is using the old version of the schema attempts can deserialize a value that uses the new version of the schema. In this case, the value retrieved from the store does not contain the age field. So upon deserialization, the age field is automatically inserted into the schema (because the reader schema requires it) and the default value is used for the newly inserted field.

As with adding fields, this has ramifications if you change your schema, and then have clients concurrently running that are using different schema versions.

1. Client v.1 creates a my.example.userInfo record, and sets the age field to 38. Then it writes that value to the store. Client v.1 is using schema version 1.

2. Client v.2 reads the record. It is using version 2 of the schema, so it is not expecting the age field. As a result, the age field is automatically stripped from the value during deserialization.

   Client v.2 modifies the name field and then writes the record back to the store. When it does this, the age field is missing from the value that it writes to the store.

3. Client v.1 reads the record again. Because the age field is missing from the record (because Client v.2 last wrote it), the age field is automatically inserted into the value, using the default of -1. This means that the value of the age field has reverted to the default, even though no client explicitly modified it.

Managing Avro Schema in the Store

This section describes how to add, change, disable and enable, and show the Avro schema in your store.

Adding Schema

Avro schema is defined in a flat-text file, and then added to the store using the command line interface. For example, suppose you have schema defined in a file called `my_schema.avsc`. Then (assuming your store is running) you start your command line interface and add the schema like this:

```
> java -jar <kvhome>/kvstore.jar runadmin -port <port> -host <host>
kv-> ddl add-schema -file my_schema.avsc
```

Note that when adding schema to the store, some error checking is performed to ensure that the schema is correctly formed. Errors are problems that must be addressed before the schema can be added to the store. Warnings are problems that should be addressed, but are

not so serious that the CLI refuses to add the schema. However, to add schema with Warnings, you must use the `-force` switch.

As of this release, the only Error that can be produced is if a field's default value does not conform to the field's type. That is, if the schema provides an integer as the default value where a string is required.

As of this release, the only Warning that can be produced is if the schema does not provide a default value for every field in the schema. Default values are required if you ever want to change (evolve) the schema. But in all other cases, the lack of a default value does no harm.

Changing Schema

To change (evolve) existing schema, use the `-evolve` flag:

```
kv-> ddl add-schema -file my_schema.avsc -evolve
```

Note that when changing schema in the store, some error checking is performed to ensure that schema evolution can be performed correctly. This error checking consists of comparing the new schema to all currently enabled versions of that schema.

This error checking can result in either Errors or Warnings. Errors are fatal problems that must be addressed before the modified schema can be added to the store. Errors represent situations where data written with an old version of the schema cannot be read by clients using a new version of the schema.

Possible errors are:

- A field is added without a default value.
- The size of a fixed type is changed.
- An enum symbol is removed.
- A union type is removed or, equivalently, a union type is changed to a non-union type and the new type is not the sole type in the old union.
- A change to a field's type (specifically to a different type name) is considered an error except when it is a type promotion, as defined by the Avro spec. And even a type promotion is a warning; see below. Another exception is changing from a non-union to a union; see below.

Warnings are problems that can be avoided using a a two-phase upgrade process. In a two-phase upgrade, all clients begin using the schema only for reading in phase I (the old schema is still used for writing), and then use the new schema for both reading and writing in phase II. Phase II may not be begun until phase I is complete; that is, no client may use the new schema for writing until all clients are using it for reading.

Possible Warnings are:

- A field is deleted in the new schema when it does not contain a default value in the old schema.

- An enum symbol is added.
- A union type is added or, equivalently, a non-union type is changed to a union that includes the original type and additional types.
- A field's type is promoted, as defined by the Avro spec. Type promotions are: int to long, float or double; long to float or double; float to double.

Disabling and Enabling Schema

You cannot delete schema, but you can disable it:

```
kv-> ddl disable-schema -name avro.MyInfo.1
```

To enable schema that has been disabled:

```
kv-> ddl enable-schema -name avro.MyInfo.1
```

Showing Schema

To see all the schemas currently enabled in your store:

```
kv-> show schemas
```

To see all schemas, including those which are currently disabled:

```
kv-> show schemas -disabled
```

Chapter 8. Avro Bindings

Once you have defined your schema (as described in Avro Schemas (page 29)), you make use of it in your Oracle NoSQL Database application in the following way:

1. Add the schema to your store. See Managing Avro Schema in the Store (page 41) for information on how to do this.
2. Identify the schema to your application.
3. Serialize and/or deserialize Oracle NoSQL Database values which use the Avro data format. You use Avro bindings to perform the serialization functions. There are different bindings available to you, each of which offers pluses and negatives.

Other than that, the mechanisms you use to read/write/delete records in the store do not change just because you are using the Avro data format with your values. Avro affects your code only where you manage your values.

The following sections describe the bindings that you use to serialize and deserialize your data. The binding that you use defines how you provide your schema to the store.

Avro Bindings Overview

There are four bindings you can use to serialize and deserialize store values that use the Avro data format. Each binding has strengths and weaknesses. The following sections go into three of the four in some detail. But, briefly, the four bindings are:

- Generic

  A generic binding is a general-purpose binding. Generic bindings identify fields to be read and written by supplying a simple string that names the field. This allows you to use the binding in a somewhat generic way, but it suffers from a lack of type safety at application compile time.

  If you are just starting out with Avro, and have no initial reason to prefer one of the other bindings, then you should start with the generic binding.

  Generic bindings are described in Generic Binding (page 45).

- Specific

  Specific bindings make use of classes that are generated from your schema specifications using an Ant tool. The generated classes allow you to manage the fields in your schema using getter and setter methods. As a programming methodology, specific schemas might represent a familiar programming style, depending on your past experiences.

  Unlike generic and JSON bindings, specific bindings require you to know in advance of compile time what all of your store schemas will be. This is because specific bindings make use of classes that are generated from your schema specifications using an Avro tool that can be called from Ant, and so there is no way for your application to dynamically discover the set of all schemas in use in the store.

Specific bindings are described in Specific Binding (page 54).

- JSON

  JSON bindings behave similarly to generic bindings, but offer less support for Avro data types. JSON bindings are most useful if you are integrating your Oracle NoSQL Database client code into a system that is JSON-oriented, because they expose JSON objects.

  JSON bindings are described in JSON Bindings (page 58).

- Raw

  Raw bindings is an advanced feature that allows you to use low-level Avro APIs to serialize and deserialize objects. Raw bindings should be used when the other three built-in bindings will not work for you, for whatever reason. Usage of Raw bindings requires a good understanding of the Avro API set. As a result, their usage is beyond the scope of this manual.

Note that you can read using one binding and write using another one. The Avro data format (which is what is actually used for the data placed into the store) is binding independent. Put another way, the bindings know how to read and write the Avro data format; the data format itself has no knowledge of the bindings.

Generic Binding

Generic bindings provide the widest support for the Avro data types. They are also flexible in that your application does not need to know the entire set of schemas in use in the store at compile time. This provides you good flexibility if your store has a constantly expanding set of schema.

The downside to generic bindings is that they do not provide compile-time type safety. Generic bindings identify fields using a string (as opposed to getter and setter methods provided by specific bindings), so it is not possible for the compiler to know, for example, whether you are using an integer where a real is expected.

Generic binding uses `AvroCatalog.getGenericBinding()` for a single schema binding, and uses `AvroCatalog.getGenericMultiBinding()` when using multiple schemas.

Using a Single Generic Schema Binding

```
{
    "type": "record",
    "name": "PersonInformation",
    "namespace": "avro",
    "fields": {"name": "ID", "type": "int"}
}
```

Further, suppose you placed that schema in a file named `PersonSchema.avsc`.

Then to use that schema, first add it to your store using the `ddl add-schema` command:

```
> java -jar <kvhome>/kvstore-m.n.o.jar runadmin -port <port> \
-host <host>
```

```
kv-> ddl add-schema -file PersonSchema.avsc
```

In your Oracle NoSQL Database client code, you must make the schema available to the code. One way to do this is to read the schema directly from the file where you created it:

```
package avro;

import java.io.File;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.Schema;

import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.GenericAvroBinding;

...

final Schema.Parser parser = new Schema.Parser();
parser.parse(new File("PersonSchema.avsc"));
```

Next, you need to make the schema available to your application:

```
final Schema personSchema =
    parser.getTypes().get("avro.PersonInformation");
```

Finally, before you can begin serializing and deserializing values that use the Avro data format, you must create a binding and then create an Avro record for that binding. In this example, we use the generic binding. But as we explain later in this chapter, there are other bindings and the generic binding may not necessarily be the best one for your purposes.

```
/**
 * Here, for the sake of brevity, we skip the necessary steps of
 * declaring and opening the store handle.
 */
final AvroCatalog catalog = store.getAvroCatalog();
final GenericAvroBinding binding =
    catalog.getGenericBinding(personSchema);
```

Once you have the binding, you need a way for your application to represent the fields in the schema, so that they can be read and written. You do this by creating an *Avro record*, which is a data structure that allows you to read and/or write the fields in the schema. (Do not confuse an Avro record, which is a handle to a binary object containing data, to an Oracle NoSQL Database record, which is a single key-value pair contained in your store. An Oracle NoSQL Database record can contain a value that uses the Avro data format. An instance of the Avro data format, in turn, is managed within your client code using an Avro record.)

Because we are using the generic binding for this example, we will use the `GenericRecord` to manage the contents of the binding.

For example, assume we performed a store read, and now we want to examine the information stored with the Oracle NoSQL Database record.

```
/**
  * Assume a store read was performed here, and resulted in a
```

```
   * ValueVersion instance called 'vv'. Then, to deserialize
   * the value in the returned record:
   */
final GenericRecord member;
final int ID;
if (vv != null) {
    /* Deserialize the the value */
    member = binding.toObject(vv.getValue());
    /* Retrieve the contents of the ID field. Because we are
     * using a generic binding, we must type cast the retrieved
     * value.
     */
     ID = (Integer) member.get("ID");
}
```

If we want to write to a field (that is, we want to serialize some data), we use the record's `put()` method. As an example, suppose we wanted to create a brand new Avro object to be written to the store. Then:

```
final GenericRecord person = new GenericData.Record(personSchema);
final int ID = 100011;
person.put("ID", ID);

/**
  * To serialize this information so that it can be written to
  * the store, use GenericBinding.toValue() as the value for the
  * store put(). That is, assuming you already have a store handle
  * and a key:
  */
store.put(key, binding.toValue(person));
```

Using Multiple Generic Schema Bindings

It is unlikely that you will use only one schema with your application. In order to use more than one schema:

1. Specify each schema individually in separate files.
2. Add all these schemas to your store as described in Managing Avro Schema in the Store (page 41).
3. Use `HashMap` to organize your schemas, and then pass that to `AvroCatalog.getGenericMultiBinding()` in order to create your binding.

For example, suppose you had the following two schemas:

```
{
 "type": "record",
 "namespace": "avro",
 "name": "PersonInfo",
 "fields": [
   { "name": "first", "type": "string" },
   { "name": "last", "type": "string" },
```

```
   { "name": "age", "type": "int" }
 ]
}


{
 "type": "record",
 "namespace": "avro",
 "name": "AnimalInfo",
 "fields": [
   { "name": "species", "type": "string"},
   { "name": "name", "type": "string"},
   { "name": "age", "type": "int"}
 ]
}
```

Then put `Avro.PersonInfo` in a file (call it `PersonSchema.avsc`) and `Avro.AnimalInfo` in a second file (`AnimalSchema.avsc`). Add these schemas to your store using the command line interface.

At this point, you could simply create one binding for each schema that you are using, but that can quickly become awkward depending on how many schemas your code is using. Instead, create multiple schemas using `HashMap` and (in this case) `AvroCatalog.getGenericMultiBinding()`. To do this, first create a `HashMap` that you use to organize your schemas:

```
package avro;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;

import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;

...

import oracle.kv.ValueVersion;
import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.GenericAvroBinding;

...

HashMap<String, Schema> schemas = new HashMap<String, Schema>();
```

Then, parse each schema and add it to the `HashMap`:

```
final Schema.Parser parser = new Schema.Parser();
```

```
Schema personSchema = parser.parse(new File("PersonSchema.avsc"));
schemas.put(personSchema.getFullName(), personSchema);

Schema animalSchema = parser.parse(new File("AnimalSchema.avsc"));
schemas.put(animalSchema.getFullName(), animalSchema);
```

Then create your binding. You will only need one, because you are using a multi binding which is capable of using multiple schemas.

```
/*
 * Store creation is skipped for brevity
 */

catalog = store.getAvroCatalog();
binding = catalog.getGenericMultiBinding(schemas);
```

To use the binding, you call `toObject()` or `put()` in the same way as you would if you were using an ordinary single-schema binding. The multi-binding is capable of determining which schema you are using, and serializing/deserializing accordingly. For example, suppose you retrieve a record that uses the `Avro.AnimalInfo` schema. Then you can deserialize as if you are using a single-schema binding:

```
/*
 * Key creation and store retrieval skipped.
 * Assume we have retrieved a ValueVersion (vv1) that
 * contains an AnimalInfo value.
 */

final GenericRecord animalObject;
if (vv1 != null) {
    animalObject = binding.toObject(vv1.getValue());
    final String species = animalObject.get("species").toString();
    final String name = animalObject.get("name").toString();
    final int age = (Integer) animalObject.get("age");

    /* Do something with the data */
}
```

You can also create a new `Avro.PersonInfo` object for placement in the store using the same binding, like this:

```
final GenericRecord personObject =
    new GenericData.Record(personSchema);
personObject.put("name", "Sam Brown");
personObject.put("age", 34);

/*
 * Key creation and store handle creation skipped
 * for brevity's sake.
 */

store.put(aKey, binding.toValue(personObject));
```

Using Embedded Records

Suppose you have a schema that looks like this:

```
{
    "type" : "record",
    "name" : "hatInventory",
    "namespace" : "avro",
    "fields" : [{"name" : "sku", "type" : "string", "default" : ""},
                  {"name" : "description",
                     "type" : {
                         "type" : "record",
                         "name" : "hatInfo",
                         "fields" : [
                                     {"name" : "style",
                                      "type" : "string",
                                      "default" : ""},

                                     {"name" : "size",
                                      "type" : "string",
                                      "default" : ""},

                                     {"name" : "color",
                                      "type" : "string",
                                      "default" : ""},

                                     {"name" : "material",
                                      "type" : "string",
                                      "default" : ""}
                            ]}
                }
    ]
}
```

In order to address the fields in the embedded record `hatInfo`, you treat it as if it is a second piece of standalone schema. You only have to parse the schema file once. You then create two schemas and two records, but only one binding. For example, to create a serialized object that uses this schema:

```
package avro;

import java.io.File;

import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.Schema;

import oracle.kv.KVStore;
import oracle.kv.Key;
import oracle.kv.ValueVersion;
import oracle.kv.avro.AvroCatalog;
```

```
import oracle.kv.avro.GenericAvroBinding;

...

// Parse our schema file
final Schema.Parser parser = new Schema.Parser();
try {
    parser.parse(new File("HatSchema.avsc"));
} catch (IOException io) {
    io.printStackTrace();
}


// Get two Schema objects. We need two because of the
// embedded record.
final Schema hatInventorySchema =
    parser.getTypes().get("avro.hatInventory");
final Schema hatInfoSchema =
    parser.getTypes().get("avro.hatInfo");

// Get two GenericRecords so we can manipulate both of
// the records in the schema
final GenericRecord hatRecord =
    new GenericData.Record(hatInventorySchema);
final GenericRecord hatInfoRecord =
    new GenericData.Record(hatInfoSchema);

// Now populate our records. Start with the
// embedded record.
hatInfoRecord.put("style", "western");
hatInfoRecord.put("size", "medium");
hatInfoRecord.put("color", "black");
hatInfoRecord.put("material", "leather");

// Now the top-level record. Notice that we
// set the embedded record as the value for the
// description field.
hatRecord.put("sku", "289163009");
hatRecord.put("description", hatInfoRecord);

// Now we need a binding. Only one is required,
// and we use the top-level schema to create it.
final AvroCatalog catalog = store.getAvroCatalog();
final GenericAvroBinding hatBinding =
    catalog.getGenericBinding(hatInventorySchema);

// Create a Key and write the value to the store.
final Key key = Key.createKey(Arrays.asList("hats", "0000000033"));
store.put(key, hatBinding.toValue(hatRecord));
```

On retrieval, you edit values of this type in the following way:

```
// Perform the retrieval
final ValueVersion vv = store.get(key);
if (vv != null) {
    // Deserialize the ValueVersion as normal
    GenericRecord hatR =
        new GenericData.Record(hatInventorySchema);
    hatR = hatBinding.toObject(vv.getValue());

    // To access the embedded record, create a GenericRecord
    // using the embedded record's schema. Then get the
    // embedded record from the field on the top-level
    // schema that contains it.
    GenericRecord hatInfoR =
        new GenericData.Record(hatInfoSchema);
    hatInfoR = (GenericRecord) hatR.get("description");

    // Finally, you can write to the top-level record and the
    // embedded record like this:

    // Modify a field on the embedded record:
    hatInfoR.put("style", "Fedora");

    // Modify the top-level record:
    hatR.put("sku", "300");
    hatR.put("description", hatInfoR);

    store.put(key, hatBinding.toValue(hatR)); }
```

Managing Generic Schemas Dynamically

A special use-case of generic bindings is that you do not necessarily know about all the schemas that will ever be used by your store at the time you write your code. That is, the use of a `HashMap` in the previous example is somewhat brittle if you are operating in an environment with a constantly growing list of schemas. In that scenario, whenever you add to the schemas in use by your store, you potentially might need to rewrite your client code to add the new schemas to the `HashMap` used by your client. Otherwise, your code could retrieve a value that uses a schema which is unknown to your code. Depending on what your code is doing, this can cause you problems.

If this is a problem for you, you can avoid it by using `AvroCatalog.getCurrentSchemas()` with `AvroCatalog.getGenericMultiBinding()` so that you do not need to build a `HashMap` of all your schemas.

For example, in the previous example we showed client code that used two known schemas. We can change the previous example to use `getCurrentSchemas()` in the following way:

```
package avro;
```

```
import java.io.File;
import java.io.IOException;
import java.util.Arrays;

import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;

...

import oracle.kv.ValueVersion;
import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.GenericAvroBinding;

...

final Schema.Parser parser = new Schema.Parser();
Schema animalSchema = parser.parse(new File("AnimalSchema.avsc"));


/*
 * We skip creating a HashMap of Schemas because
 * it is not needed.
 */


/*
 * Store creation is skipped for brevity
 */

catalog = store.getAvroCatalog();
binding = catalog.getGenericMultiBinding(catalog.getCurrentSchemas());
```

If we then perform a read on the store, there is the possibility that we will retrieve an object that uses a schema which was not in use when our binding was created. (This is particularly true for long-running client code). To handle this problem, we catch `SchemaNotAllowedException`.

```
/*
 * Key creation and store retrieval skipped.
 */

final GenericRecord animalObject;
if (vv1 != null) {
    try {
        animalObject = binding.toObject(vv1.getValue());
    } catch (SchemaNotAllowedException e) {
        // Take some action here. Potentially you could
        // recreate your binding, and then retry the
        // deserialization process
```

```
    }

    /*
     * Do something with the data. If your client code is
     * using more than one schema, you can identify which
     * schema the retrieved value is using by testing
     * the schema name. That is:
     *
     * String sName = animalObject.getSchema().getFullName()
     * if (sName.equals("avro.animalInfo")) { ... }
     */
}
```

Specific Binding

The `SpecificAvroBinding` interface represents values as instances of a generated Avro class which implements `SpecificRecord`. A single schema binding is created using `AvroCatalog.getSpecificBinding()`. A multiple schema binding is created using `AvroCatalog.getSpecificMultiBinding()`.

Avro-specific classes provide type safety and ease of use. Properties are accessed through getter/setter methods with parameters and return values of the correct type, as defined in the schema.

Further, when using Avro-specific classes, the client application does not need to be aware of the Avro schema at runtime. The generated class has an internal reference to its associated schema, and this schema is used by the binding, so the application does not need to explicitly supply a schema. Schemas are supplied at build time, when the source code for the specific class is generated.

The disadvantage to using specific bindings is that the set of specific classes is fixed at build time. Applications that wish to treat values generically, or dynamically based on the schema, should use a generic or JSON binding instead.

Generating Specific Avro Classes

When you use a specific binding, you provide your schema to your application using a generated class. You can do this using the `org.apache.avro.compiler.specific.SchemaTask` tool.

An example Ant file that uses `SchemaTask` can be found in `KVHOME/examples/avro/generate-specific.xml`. It automatically downloads all library dependencies required to generate the Avro class, and then creates generated classes for all schema found in the local directory. The example assumes your schema is contained in flat text files that use the `avsc` suffix.

For example, suppose you had a schema defined in a file named `my-schema.avsc`. Then to generate an Avro-specific class, place `generate-specific.xml` and `my-schema.avsc` in the same directory and run:

```
ant -f generate-specific.xml
```

After Ant downloads all the necessary library dependencies, it will create whatever generated classes are required by the contents of `my-schema.avsc`. The generated classes are named after the `name` field in the Avro schema. So if you had the (trivial) schema:

```
{
    "type": "record",
    "name": "MyInfoString",
    "namespace": "avro",
    "fields": [
        {"name": "ID", "type": "int"}
    ]
}
```

the generated class name would be `MyInfoString.java`. It contains getter and setter methods that you can use to access the fields in the schema. For example, a generated class for the above schema would contain the fields `MyInfoString.setID()` and `MyInfoString.getID()`.

Using Avro-specific Bindings

To use a schema encapsulated in a generated Avro-specific class, you first provide the schema to the store as described in Managing Avro Schema in the Store (page 41).

After that, you access the schema using the generated class. To do this, you need handles to the store, an Avro catalog, and a specific binding:

```
package avro;

import oracle.kv.KVStore;
import oracle.kv.ValueVersion;
import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.SpecificAvroBinding;

...


private final KVStore store;
private final AvroCatalog catalog;
private final SpecificAvroBinding<MyInfoString> binding;

...

/* store configuration and open omitted for brevity */

...

catalog = store.getAvroCatalog();
binding = catalog.getSpecificBinding(MyInfoString.class);
```

Having done that, you can use the binding to serialize and deserialize value objects that conform to the generated class' schema. For example, suppose you performed a store `get()` and retrieved a `ValueVersion`. Then you can access its information in the following way:

```
final MyInfoString member;
final int ID;

member = binding.toObject(valueVersion.getValue());

System.out.println("ID: " + member.getID());
```

To serialize the object for placement in the store, you use the generated class' setter method, as well as the binding's `toValue()` method:

```
member.setID(2045);
/* key creation ommitted */
store.put(key, binding.toValue(member));
```

Using Multiple Avro-specific Bindings

If your client code needs to use multiple specific bindings, then you use `AvroCatalog.getMultiSpecificBinding()` to support the multiple schemas. For example, suppose your client code required the following two schemas:

```
{
 "type": "record",
 "namespace": "avro",
 "name": "PersonInfo",
 "fields": [
   { "name": "first", "type": "string" },
   { "name": "last", "type": "string" },
   { "name": "age", "type": "int" }
 ]
}


{
 "type": "record",
 "namespace": "avro",
 "name": "AnimalInfo",
 "fields": [
   { "name": "species", "type": "string"},
   { "name": "name", "type": "string"},
   { "name": "age", "type": "int"}
 ]
}
```

Then put `Avro.PersonInfo` in a file (call it `PersonSchema.avsc`) and `Avro.AnimalInfo` in a second file (`AnimalSchema.avsc`). Add these schemas to your store using the command line interface.

Next, generate your specific schema classes using:

```
ant -f generate-specific.xml
```

as discussed in Generating Specific Avro Classes (page 54). This results in two generated classes: `PersonInfo.java` and `AnimalInfo.java`. To use these classes in your client code, you can use a single binding, which you create using `AvroCatalog.getSpecificMultiBinding()`. Note that to do this, you must use `org.apache.avro.specific.SpecificRecord`:

```
package avro;

import java.util.Arrays;

import oracle.kv.Key;
import oracle.kv.ValueVersion;
import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.SpecificAvroBinding;

import org.apache.avro.specific.SpecificRecord;

...

private final KVStore store;
private final AvroCatalog catalog;
private final SpecificAvroBinding<SpecificRecord> binding;

/*
 * Store creation is skipped for the sake of brevity.
 */

...

catalog = store.getAvroCatalog();
binding = catalog.getSpecificMultiBinding();
```

You can then create `PersonInfo` and `AnimalInfo` objects, serialize them, and write them to the store, like this:

```
final Key key1 = Key.createKey(Arrays.asList("person", "11"));
final Key key2 = Key.createKey(Arrays.asList("animal", "11"));

final PersonInfo pi = new PersonInfo();
pi.setFirst("Jack");
pi.setLast("Smith");
pi.setAge(38);
store.put(key1, binding.toValue(pi));

final AnimalInfo ai = new AnimalInfo();
ai.setSpecies("Dog");
ai.setName("Bowzer");
ai.setAge(6);
store.put(key2, binding.toValue(ai));
```

Retrieval of the stored objects is performed with a normal store `get()`. However, to deserialize the retrieved objects you must identify the object's type. You can do this using either the Java `instanceof` operator, or by examining the schema's name, as follows:

```
final ValueVersion vv = store.get(someKey);
if (vv != null) {
    SpecificRecord sr = binding.toObject(vv.getValue());
    if (sr.getSchema().getFullName().equals("avro.PersonInfo")) {
        PersonInfo o = (PersonInfo) sr;
        /*
         * The object is now deserialized. You can access the object's
         * data using the specific class' getXXX() methods. For example,
         * o.getFirst().
         */
    } else {
        AnimalInfo o = (AnimalInfo) sr;
        /*
         * The object is now deserialized. You can access the object's
         * data using the specific class' getXXX() methods. For example,
         * o.getSpecies().
         */
    }
```

JSON Bindings

The `JsonAvroBinding` interface represents values as instances of `JsonRecord`. A single schema binding is created using `AvroCatalog.getJsonBinding()`. A multiple schema binding is created using `AvroCatalog.getJsonMultiBinding()`.

The most important reason to use a JSON binding is for interoperability with other components or external systems that use JSON objects. This is because JSON bindings expose JSON objects, which can be managed with the Jackson API; a popular API used to manage JSON records.

Like generic bindings, a JSON binding treats values generically. And, like generic bindings, the schemas used in the application need not be fixed at build time if the schemas are treated dynamically.

However, unlike `GenericRecord`, certain Avro data types are not represented conveniently using the JSON syntax:

- Avro `int` and `long` are both represented as a JSON `integer`.

  ## Note

  To avoid type conversion errors, it is safest to always define integer fields in the Avro schema using type `long` rather than `int` when using this binding.

- Avro `float` and `double` are both represented as a JSON `number`.

Note

Because Jackson represents floating point numbers as a Java `Double`, you must define floating point fields in the Avro schema using type `double` rather than `float` when using this binding.

- Avro `record` and `map` are represented as a JSON `object`.
- Avro `bytes` and `fixed` are both represented as a JSON `string`, using Unicode escape syntax.

Note

Characters greater than 0xFF are invalid because they cannot be translated to bytes without loss of information.

- An Avro `enum` is represented as a JSON `string`.
- Avro unions have a special JSON representation. .

For this reason, applications that use the JSON binding should limit the data types used in their Avro schemas, and should treat the above data types carefully.

Like `GenericRecord`, a JSON object does not provide type safety. It can be error prone, because fields are accessed by string name and so data types cannot be checked at compile time.

Using Avro JSON Bindings

To use schema encapsulated in a generated Avro-specific class, you first provide the schema to the store as described in Managing Avro Schema in the Store (page 41).

In your Oracle NoSQL Database client code, you must make the schema available to the code. Do this by reading the schema directly from the file where you created it. For example, suppose you had the following schema defined in a file named `my-schema.avsc`:

```
{
    "type": "record",
    "name": "MyInfo",
    "namespace": "avro",
    "fields": [
        {"name": "ID", "type": "int"}
    ]
}
```

Then to read that schema into your client code:

```
package avro;

import java.io.File;
```

```
import org.apache.avro.Schema;
import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.node.JsonNodeFactory;
import org.codehaus.jackson.node.ObjectNode;

import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.JsonAvroBinding;
import oracle.kv.avro.JsonRecord;

...

final Schema.Parser parser = new Schema.Parser();
parser.parse(new File("my-schema.avsc"));
final Schema myInfoSchema =
    parser.getTypes().get("avro.MyInfo");
```

Before you can begin serializing and deserializing values that use the Avro data format, you must create a JSON binding and then create an Avro record for that binding.

```
/**
 * For the sake of brevity, we skip the necessary steps of
 * declaring and opening the store handle.
 */
final AvroCatalog catalog = store.getAvroCatalog();
final JsonAvroBinding binding =
    catalog.getJsonBinding(myInfoSchema);
```

Once you have the binding, you need a way for your application to represent the fields in the schema, so that they can be read and written. You do this by creating a *Json record* and from there an *Object Node*, which is a data structure that allows you to read and/or write the fields in the schema.

For example, assume we performed a store read, and now we want to examine the information stored with the Oracle NoSQL Database record.

```
/**
  * Assume a store read was performed here, and resulted in a
  * ValueVersion instance called 'vv'. Then, to deserialize
  * the value in the returned record:
  */
final JsonRecord record = binding.toObject(vv.getValue());
final ObjectNode member = (ObjectNode) record.getJsonNode();
/* Retrieve the contents of the ID field. Because we are
 * using a generic binding, we must type cast the retrieved
 * value.
 */
final int ID = member.get("ID").getIntValue();
```

If we want to write to a field (that is, we want to serialize some data), we use the member's `put()` method. As an example, suppose we wanted to create a brand new object to be written to the store. Then:

```
final ObjectNode member2 = JsonNodeFactory.instance.objectNode();
member2.put("ID", 100011);
final JsonRecord record2 = new JsonRecord(member2, myInfoSchema);
...
/**
  * Assume the creation of a store key here.
 */
...
store.put(key, binding.toValue(record2));
```

Using a JSON Binding with a JSON Record

The most common usage of the JSON binding is to store information that is already marked up with JSON. A typical example would be data collected from a web server in JSON format that you then want to place in the store.

Because Oracle NoSQL Database's Avro implementation relies on the Jackson API, which is a common API used to parse JSON records, everything you need is already in place to put a JSON record into the store.

Suppose you had a JSON record that looked like this:

```
{
    "name": {
        "first": "Percival",
        "last":  "Lowell"
    },
    "age": 156,
    "address": {
        "street": "Mars Hill Rd",
        "city": "Flagstaff",
        "state": "AZ",
        "zip": 86001
    }
}
```

To support records of this type, you need a schema definition:

```
{
    "type": "record",
    "name": "MemberInfo",
    "namespace": "avro",
    "fields": [
        {"name": "name", "type": {
            "type": "record",
            "name": "FullName",
            "fields": [
                {"name": "first", "type": "string"},
                {"name": "last", "type": "string"}
            ]
        }},
```

```
        {"name": "age", "type": "int"},
        {"name": "address", "type": {
            "type": "record",
            "name": "Address",
            "fields": [
                {"name": "street", "type": "string"},
                {"name": "city", "type": "string"},
                {"name": "state", "type": "string"},
                {"name": "zip", "type": "int"}
            ]
        }}
    ]
}
```

Assuming that you added that schema to a file called `MemberInfo.avsc`, you add it to the store using the CLI (using the `ddl add-schema` command), and then create your JSON binding in the same way as shown in the previous example.

(Note that one additional class is required for this example: `ObjectMapper`. In addition, we use `java.io.BufferedReader` and `java.io.FileReader` to support reading the JSON record from disk, which is something that you probably would not do in production client code.)

```
package avro;

import java.io.BufferedReader;
import java.io.File;
import org.apache.avro.Schema;
import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.JsonNodeFactory;
import org.codehaus.jackson.node.ObjectNode;

import oracle.kv.avro.AvroCatalog;
import oracle.kv.avro.JsonAvroBinding;
import oracle.kv.avro.JsonRecord;

import oracle.kv.Key;
import oracle.kv.Value;

...

final Schema.Parser parser = new Schema.Parser();
parser.parse(new File("MemberInfo.avsc"));
final Schema memberInfoSchema =
    parser.getTypes().get("avro.MemberInfo");

...

/**
 * For the sake of brevity, we skip the necessary steps of
```

```
 * declaring and opening the store handle.
 */

...

final AvroCatalog catalog = store.getAvroCatalog();
final JsonAvroBinding binding =
    catalog.getJsonBinding(memberInfoSchema);
```

The next step is to read the JSON record into your application. Typically, this would occur over the network, but for simplicity, we show the record being read from a file on disk:

```
try {
    final String jsonText =
        readFile(new File("MemberInfoRecord.json"));
```

Next, we use the Jackson API to parse the text so as to create a JSON object:

```
    final ObjectMapper jsonMapper = new ObjectMapper();

    final JsonNode jsonObject = jsonMapper.readTree(jsonText);
```

Finally, we write the JSON object to a JSON record, which we then use to create an Oracle NoSQL Database `Value`.

```
    final JsonRecord jsonRecord =
        new JsonRecord(jsonObject, memberInfoSchema);

    final Value value = binding.toValue(jsonRecord);
```

The final step is to the write the object to the store in the usual way:

```
    store.put(Key.fromString("/any/old/key"), value);
} catch (IOException io) {
    io.printStackTrace();
}
```

To be complete, the code for the `readFile()` method used in this example is:

```
private String readFile(File f) throws IOException {
    final BufferedReader r = new BufferedReader(new FileReader(f));
    final StringBuilder buf = new StringBuilder(1000);
    String line;
    while ((line = r.readLine()) != null) {
        buf.append(line);
        buf.append("\n");
    }
    return buf.toString();
}
```

Chapter 9. Key Ranges and Depth for Multi-Key Operations

When performing multi-key operations (for example, `KVStore.multiGet()`, `KVStore.multiDelete()`, `KVStore.storeIterator()`), you can limit the records that are operated upon by specifying key ranges and depth. Key ranges allow you to identify a subset of keys to use out of a matching set. Depth allows you to specify how many children you want the multi-key operation to use.

Specifying Subranges

When performing multi-key operations in the store, you can specify a range of records to operate upon. You do this using the `KeyRange` class. This class defines a range of `String` values for the key components immediately following a key that is used in a multiple get operation.

For example, suppose you were using the following keys:

```
/Smith/Bob/-/birthdate
/Smith/Bob/-/phonenumber
/Smith/Bob/-/image
/Smith/Bob/-/userID
/Smith/Patricia/-/birthdate
/Smith/Patricia/-/phonenumber
/Smith/Patricia/-/image
/Smith/Patricia/-/userID
/Smith/Richard/-/birthdate
/Smith/Richard/-/phonenumber
/Smith/Richard/-/image
/Smith/Richard/-/userID
/Wong/Bill/-/birthdate
/Wong/Bill/-/phonenumber
/Wong/Bill/-/image
/Wong/Bill/-/userID
```

Given this, you could perform operations for all the records related to users Bob Smith and Patricia Smith by constructing a `KeyRange`. When you do this, you must identify the key components that define the upper and lower bounds of the range. You must also identify if the key components that you provide are inclusive or exclusive.

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.

```
package kvstore.basicExample;

import oracle.kv.KeyRange;

...
```

```
KeyRange kr = new KeyRange("Bob", true, "Patricia", true);
```

You then use the `KeyRange` instance when you perform your multi-key operation. For example, suppose you were retrieving records from your store using `KVStore.storeIterator()`:

```
package kvstore.basicExample;

...

import oracle.kv.Direction;
import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.KeyRange;
import oracle.kv.KeyValueVersion;
import oracle.kv.RequestTimeoutException;

import java.util.ArrayList;
import java.util.Iterator;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");

// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

KeyRange kr = new KeyRange("Bob", true, "Patricia", true);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.


try {
    Iterator<KeyValueVersion> i =
        kvstore.storeIterator(Direction.FORWARD, 0,
                              myKey, kr, null);
    while (i.hasNext()) {
        Value v = i.next().getValue();
        // Do some work with the Value here
    }
} catch (RequestTimeoutException re) {
    // The operation was not completed within the
    // timeout value
}
```

Specifying Depth

When performing multi-key operations in the store, you can specify a depth of records to operate upon. That is, you can indicate whether you want to operate upon:

- The specified key and all its children.
- The specified key and its most immediate children.
- Only the immediate children of the specified key. (The specified key is omitted.)
- All of the children of the specified key. (The specified key is omitted.)

By default, multi-key operations operate upon the specified key and all of its children. To limit the operation to something else, such as just the key's immediate children, specify `Depth.CHILDREN_ONLY` to the operation's `Depth` parameter.

For example, suppose you were using the following keys:

```
/Products/Hats/-/baseball
/Products/Hats/-/baseball/longbill
/Products/Hats/-/baseball/longbill/blue
/Products/Hats/-/baseball/longbill/red
/Products/Hats/-/baseball/shortbill
/Products/Hats/-/baseball/shortbill/blue
/Products/Hats/-/baseball/shortbill/red
/Products/Hats/-/western
/Products/Hats/-/western/felt
/Products/Hats/-/western/felt/black
/Products/Hats/-/western/felt/gray
/Products/Hats/-/western/leather
/Products/Hats/-/western/leather/black
/Products/Hats/-/western/leather/gray
```

Further, suppose you wanted to retrieve just these records:

```
/Products/Hats/-/baseball
/Products/Hats/-/western
```

Then you could do this using `KVStore.multiGet()` with the appropriate `Depth` argument.

```
package kvstore.basicExample;

...

import oracle.kv.Depth;
import oracle.kv.Key;
import oracle.kv.RequestTimeoutException;
import oracle.kv.Value;
import oracle.kv.ValueVersion;

import java.util.ArrayList;
```

```
import java.util.Iterator;
import java.util.SortedMap;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Product");
majorComponents.add("Hats");

// Create the retrieval key
Key myKey = Key.createKey(majorComponents);

// Now retrieve the records. Note that we do not show the creation of
// the kvstore handle here.

try {
    SortedMap<Key, ValueVersion> myRecords = null;

    myRecords = kvstore.multiGet(myKey,
                                 null,
                                 Depth.CHILDREN_ONLY);
} catch (RequestTimeoutException re) {
    // The operation was not completed within the
    // timeout value
}
```

Chapter 10. Using Versions

When a record (that is, a key-value pair) 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 the record to the store (for example, `KVStore.put()`). The version information is also returned by methods that retrieve records 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 record's value has not changed. This is particularly interesting in an application where there can be multiple threads or processes simultaneously operating on the record. In this case, read the record, retrieving 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. You use `KVStore.putIfVersion()` or `KVStore.deleteIfVersion()` to guarantee this.

2. When a client reads a value 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 value as a consistency parameter to the read operation. For more information on using consistency, see Consistency Guarantees (page 70).

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

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

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.ValueVersion;
import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

minorComponents.add("phonenumber");
```

```
// Create the key
Key myKey = Key.createKey(majorComponents, minorComponents);

// Now retrieve the record. Note that we do not show the creation of
// the kvstore handle here.

ValueVersion vv = kvstore.get(myKey);
Value value = vv.getValue();
Version version = vv.getVersion();

...

//////////////////////////////////////////////////////
////////////// Do work on the value 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 record was changed elsewhere.
// In this case, you could retry the entire get/putIfVersion
// operation.
Version newVersion = kvstore.putIfVersion(myKey, value, version);
if (newVersion == null) {
    // Unsuccessful. Someone else probably modified the record.
}
```

Chapter 11. Consistency Guarantees

The KV store is built from some number of computers (called nodes) that are working together using a network. Every record in your store is first written to a master node. The master node then copies that record to other nodes in the store.

Because of the relatively slow performance of networks, 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 record on Node A to be different from the same record on Node B.

When there is a high likelihood that a record stored on one node is identical to the same record 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 record on one node differs in some way from the same record store 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 write 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.

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 using the `Consistency` parameter on the `KVStore` method 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 Pre-Defined 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 record(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 in the store), then you will probably see performance problems sooner rather than later. 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.

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.Key;
import oracle.kv.Value;
import oracle.kv.ValueVersion;

import java.util.ArrayList;

...

ArrayList<String> majorComponents = new ArrayList<String>();
```

```
...

// Define the major path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

// Create the key
Key myKey = Key.createKey(majorComponents);

// Now retrieve the record. Note that we do not show the creation of
// the kvstore handle here.

try {
    ValueVersion vv = kvstore.get(myKey,
                        Consistency.ABSOLUTE,
                        0,     // Timeout parameter.
                               // 0 means use the default.
                        null); // Timeout units. Null because
                               // the Timeout is 0.

    Value v = vv.getValue();
    /*
     * From here, deserialize using your Avro binding.
     */
} 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. These type of consistency policies matter only for read operations, because replica nodes can only service read operations.

When servicing a read operation, 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 fragments perform a store write, collect the version information, then use it to construct a version-based consistency policy. In this example, assume we are using a generic Avro binding to store some person information.

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.Version;
import java.util.ArrayList;

import org.apache.avro.Schema;
import oracle.kv.avro.GenericAvroBinding;
import oracle.kv.avro.GenericRecord;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

// Create the key
Key myKey = Key.createKey(majorComponents);
```

```
...

// Binding and schema creation omitted

...

final GenericRecord person = new GenericData.Record(personSchema);
person.put("ID", 100011);
person.put("FamiliarName", "Bob");
person.put("Surname", "Smith");
person.put("PrimaryPhone", "408 555 5555");

Value myValue = binding.toValue(person);

// Now put the record. Note that we do not show the creation of the
// kvstore handle here.

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.Key;
import oracle.kv.Value;
import oracle.kv.ValueVersion;
import oracle.kv.Version;

import org.apache.avro.Schema;
import oracle.kv.avro.GenericAvroBinding;
import oracle.kv.avro.GenericRecord;

import java.util.ArrayList;
import java.util.concurrent.TimeUnit;

...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");
```

```
// Create the key
Key myKey = Key.createKey(majorComponents);

// Create the consistency policy
Consistency.Version versionConsistency =
        new Consistency.Version(matchVersion,
                                200,
                                TimeUnit.NANOSECONDS);

// Now retrieve the record. Note that we do not show the creation of
// the kvstore handle here.

try {
    ValueVersion vv = kvstore.get(myKey,
                          versionConsistency,
                          0,     // Timeout parameter.
                                 // 0 means use the default.
                          null); // Timeout units. Null because
                                 // the Timeout is 0.

    // Deserialize with our generic avro binding
    // (creation of this binding is not shown).

    final GenericRecord member = binding.toObject(vv.getValue());

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

Chapter 12. 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 replication group. It is possible to ask the master node to wait for acknowledgments from its replicas before considering the operation complete.

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

The sum total of all this write activity is called the *durability guarantee*. That is, a durability guarantee 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 within the store. 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 to set the durability guarantee for that data to the lowest possible level.

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.

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 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.

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, or acknowledgments from all replica nodes.

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 replication group. Remember that your store has more than one replication group, 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. That is, if the replication group has 5 replica nodes, 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 write to be placed on 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 write is placed into the host's in-memory cache, but the master node does not wait for that operation 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 write operation is placed into 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 write operation 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 write operation eventually makes it 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 overwrite 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.Key;
import oracle.kv.RequestTimeoutException;
import oracle.kv.Value;
import java.util.ArrayList;

import org.apache.avro.Schema;
import oracle.kv.avro.GenericAvroBinding;
import oracle.kv.avro.GenericRecord;
```

```
...

ArrayList<String> majorComponents = new ArrayList<String>();

...

// Define the major and minor path components for the key
majorComponents.add("Smith");
majorComponents.add("Bob");

// Create the key
Key myKey = Key.createKey(majorComponents);

...

// Binding and schema creation omitted

...

final GenericRecord person = new GenericData.Record(personSchema);
person.put("ID", 100011);
person.put("FamiliarName", "Bob");
person.put("Surname", "Smith");
person.put("PrimaryPhone", "408 555 5555");

Value myValue = binding.toValue(person);

// Create the special durability policy
Durability durability =
    new Durability(Durability.SyncPolicy.NO_SYNC, // Master sync
                   Durability.SyncPolicy.NO_SYNC, // Replica sync
                   Durability.ReplicaAckPolicy.NONE);

// Now put the record. Note that we do not show the creation of the
// kvstore handle here.
try {
    kvstore.put(myKey, myValue,
                null,       // ReturnValueVersion is null because
                            // we aren't using it.
                durability, // The per-operation durability
                0,          // Use the default request timeout
                null);      // Use the default timeunit value
} catch (DurabilityException de) {
    // The durability guarantee was not met
} catch (RequestTimeoutException re) {
    // The operation was not completed within the
    // timeout value
}
```

Chapter 13. Executing a Sequence of Operations

You can execute a sequence of write operations as a single atomic unit so long as all the records that you are operating upon share the same major path components. 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 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 store puts and deletes, but you cannot perform store gets when using sequences.

When using a sequence of operations:

- All of the keys in use by the sequence must share the same major path components.
- 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.
- You cannot create two or more operations that operate on the same key. Doing so results in an exception, and the entire operation is aborted.

The rest of this chapter shows how to use `OperationFactory` and `KVStore.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`, or a `ConsistencyException`. 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 key that does not exist). If this occurs AND you specified `true` for the `abortIfUnsuccessful` parameter when the operation was created using the OperationFactory, then an `OperationExecutionException` is thrown. This exception contains information about the failed operation.

Creating a Sequence

You create a sequence by using the `OperationFactory` class to create `Operation` class instances, each of which represents exactly one operation in the store. You obtain an instance of `OperationFactory` by using `KVStore.getOperationFactory()`.

For example, suppose you are using the following keys:

```
/Products/Hats/-/baseball
/Products/Hats/-/baseball/longbill
/Products/Hats/-/baseball/longbill/blue
/Products/Hats/-/baseball/longbill/red
/Products/Hats/-/baseball/shortbill
/Products/Hats/-/baseball/shortbill/blue
/Products/Hats/-/baseball/shortbill/red
/Products/Hats/-/western
/Products/Hats/-/western/felt
/Products/Hats/-/western/felt/black
/Products/Hats/-/western/felt/gray
/Products/Hats/-/western/leather
/Products/Hats/-/western/leather/black
/Products/Hats/-/western/leather/gray
```

And further suppose each of the following records has some information (such as a price refresh date) that you want to read and update in such a fashion as to make sure that the information is consistent for all of the records:

```
/Products/Hats/-/western
/Products/Hats/-/western/felt
/Products/Hats/-/western/leather
```

The you can create a sequence in the following way:

```
package kvstore.basicExample;

...

import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.Operation;
import oracle.kv.OperationFactory;
import java.util.ArrayList;

import org.apache.avro.Schema;
import oracle.kv.avro.GenericAvroBinding;
import oracle.kv.avro.GenericRecord;

...

// Get the operation factory. Note that we do not show the
// creation of the kvstore handle here.
```

```
OperationFactory of = kvstore.getOperationFactory();

// We need a List to hold the operations in the
// sequence.
ArrayList<Operation> opList = new ArrayList<Operation>();

...

ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents1 = new ArrayList<String>();
ArrayList<String> minorComponents2 = new ArrayList<String>();
ArrayList<String> minorComponents3 = new ArrayList<String>();

...

// Define the major and minor path components for our keys
majorComponents.add("Products");
majorComponents.add("Hats");

minorComponents1.add("western");
minorComponents2.add("western");
minorComponents2.add("felt");
minorComponents3.add("western");
minorComponents3.add("leather");

// Create the three keys that we will need
Key key1 = Key.createKey(majorComponents, minorComponents1);
Key key2 = Key.createKey(majorComponents, minorComponents2);
Key key3 = Key.createKey(majorComponents, minorComponents3);

...

// Binding and schema creation omitted

...

final GenericRecord hat1 = new GenericData.Record(hatSchema);
hat1.put("randomHatData", "someRandomData");
final Value value1 = binding.toValue(hat1);

final GenericRecord hat2 = new GenericData.Record(hatSchema);
hat2.put("randomHatData", "someMoreRandomData");
final Value value2 = binding.toValue(hat2);

final GenericRecord hat3 = new GenericData.Record(hatSchema);
hat3.put("randomHatData", "someMoreRandomData");
final Value value3 = binding.toValue(hat3);

...
```

```
// Here we would perform whatever actions we need to create
// our record values. We won't show how the values get created,
// but assume it results in three Value objects: value1, value2,
// and value3.

...

// Now create our list of operations for the key pairs
// key1/value1, key2/value2, and key3/value3. In this
// trivial example we will put store all three records
// in a single atomic operation.


opList.add(of.createPut(key1, value1));
opList.add(of.createPut(key2, value2));
opList.add(of.createPut(key3, value3));
```

Note in the above example that we create three unique keys that differ only in their minor path components. If the major path components were different for any of the three keys, we could not successfully execute the sequence.

Executing a Sequence

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

```
package kvstore.basicExample;

...

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

...

try {
    kvstore.execute(opList);
} 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 a OperationResult object, which you can
    // use to troubleshoot the cause of the execution
    // exception.
} catch (DurabilityException de) {
    // The durability guarantee could not be met.
```

```
} catch (IllegalArgumentException iae) {
    // An operation in the list was null or empty.

    // Or at least one operation operates on a key
    // with a major path component that is different
    // than the others.

    // Or more than one operation uses the same key.
} 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.

A richer form of `KVStore.execute()` is available. It allows you to specify:

- The list of operations.
- 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 example:

```
package kvstore.basicExample;

...

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

import java.util.concurrent.TimeUnit;

...

Durability durability =
    new Durability(Durability.SyncPolicy.NO_SYNC, // Master sync
```

```
                    Durability.SyncPolicy.NO_SYNC, // Replica sync
                    Durability.ReplicaAckPolicy.NONE);

try {
    kvstore.execute(opList,
                     durability,
                     2000,
                     TimeUnit.MILLISECONDS);
} 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 a OperationResult object, which you can
    // use to troubleshoot the cause of the execution
    // exception.
} catch (DurabilityException de) {
    // The durability guarantee could not be met.
} catch (IllegalArgumentException iae) {
    // An operation in the list was null or empty.

    // Or at least one operation operates on a key
    // with a major path component that is different
    // than the others.

    // Or more than one operation uses the same key.
} catch (RequestTimeoutException rte) {
    // The operation was not completed inside of the
    // default request timeout limit.
} catch (FaultException fe) {
    // A generic error occurred
}
```

Chapter 14. Using Large Objects

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

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

LOB Keys

Keys associated with LOBs must have a trailing suffix string at the end of their final key component. By default, this suffix is `.lob`, but this may be defined using `KVStoreConfig.setLOBSuffix()`.

For example, a LOB key used for an image file might be:

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

Note that all of the LOB APIS verify that the key used to access LOBs meets this trailing suffix requirement. They throw an `IllegalArgumentException` if the verification fails. This requirement permits non-LOB methods to check for inadvertent modifications to LOB objects.

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

- All non-LOB write operations check for the absence of the LOB suffix as part of the other key validity checks. If the check fails it will result in an `IllegalArgumentException`.
- All non-LOB read operations return the associated opaque value used internally to construct a LOB stream.
- All LOB write and read operations in this interface check for the presence of the LOB suffix. If the check fails it will result in an `IllegalArgumentException`.

LOB APIs

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

The LOB APIs are:

- `KVStore.deleteLOB()`

  Deletes a LOB from the store.

- `KVStore.getLOB()`

  Retrieves a LOB from the store.

- `KVStore.putLOB()`

  `KVStore.putLOBIfAbsent()`

  `KVStore.putLOBIfPresent()`

  Writes a LOB to the store.

The methods used to read and insert LOBs are not atomic. Relaxing the atomicity requirement permits distribution of chunks across the entire store. It is the application's responsibility to coordinate operations on a LOB. The LOB implementation will make a good faith effort to detect concurrent modification of a LOB and throw `ConcurrentModificationException` when it detects such concurrency conflicts but does not guarantee that it will detect all such conflicts. The safe course of action upon encountering this exception is to delete the LOB and replace it with a new value after fixing the application level coordination issue that provoked the exception.

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

LOB Example

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

Also, this example only catches the bare minimum `IOException`. In a real-world example, you should at least catch and decide what to do with `PartialLOBException`, which can be thrown by `KVStore.getLOB()`. It indicates that you have only retrieved a portion of the requested object. Also, `RequestTimeoutException` can be thrown on the put operation if each chunk is not successfully read or written within the amount of time.

```
package kvstore.lobExample;

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

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

...

// Skipping configuring and opening the store

...

// Use key "/test/lob/1.lob" to store the jpeg object.
final Key key = Key.createKey(Arrays.asList("test", "lob", "1.lob"));

...

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

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


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

 InputStream stream = istreamVersion.getInputStream();
```

```
int byteCount = 0;
while (stream.read() != -1) {
    byteCount++;
}
System.out.println(byteCount);
```