Oracle

Getting Started with NoSQL Database Table Java Driver

12c Release 1

Library Version 12.1.3.5

ORACLE

NOSQL DATABASE

Legal Notice

Published 12/3/2015

Table of Contents

Preface vi
Conventions Used in This Book vi
1. Developing for Oracle NoSQL Database 1
The KVStore Handle 2
The KVStoreConfig Class 2
Using the Authentication APIs 3
Configuring SSL 4
Identifying the Trust Store 4
Setting the SSL Transport Property 4
Authentication using LoginCredentials 5
Renewing Expired Login Credentials 7
Authentication using Kerberos 9
Authentication using Kerberos and JAAS 11
Unauthorized Access 13
2. Introduction to Oracle KVLite 14
Starting KVLite 14
Stopping and Restarting KVLite 15
Verifying the Installation 15
kvlite Utility Command Line Parameter Options 16
3. Introducing Oracle NoSQL Database Tables and Indexes 18
Defining Tables 18
Executing DDL Statements using the CLI 20
Supported Table Data Types 20
Record Fields 21
Defining Child Tables 22
Table Evolution 23
Creating Indexes 23
Indexable Field Types 26
4. Primary and Shard Key Design 27
Primary Keys 27
Data Type Limitations 28
Partial Primary Keys 28
Shard Keys 28
Row Data 29
5. Writing and Deleting Table Rows 32
Write Exceptions 32
Writing Rows to a Table in the Store 32
Writing Rows to a Child Table 34
Other put Operations 35
Deleting Rows from the Store 35
Using multiDelete() 36
6. Reading Table Rows 39
Read Exceptions 39
Retrieving a Single Row 40
Retrieve a Child Table 41
Using multiGet() 42

Iterating over Table Rows 44
Specifying Field Ranges 47
Iterating with Nested Tables 49
Reading Indexes 54
Parallel Scans 56
Bulk Get Operations 58
7. Using Data Types 62
Using Arrays 62
Using Binary 63
Using Enums 64
Using Fixed Binary 65
Using Maps 67
Using Embedded Records 68
8. Indexing Non-Scalar Data Types 70
Indexing Arrays 70
Indexing Maps 71
Indexing by Map Keys 72
Indexing by Map Values 73
Indexing by a Specific Map Key Name 75
Indexing by Map Key and Value 77
Indexing Embedded Records 78
9. Using Row Versions 81
10. Consistency Guarantees 83
Specifying Consistency Policies 83
Using Simple Consistency 84
Using Time-Based Consistency 85
Using Version-Based Consistency 86
11. Durability Guarantees 90
Setting Acknowledgment-Based Durability Policies 90
Setting Synchronization-Based Durability Policies 91
Setting Durability Guarantees 92
12. Executing a Sequence of Operations 95
Sequence Errors 95
Creating a Sequence 95
Executing a Sequence 98
A. Table Data Definition Language Overview 100
Name Constraints 100
DDL Comments 101
CREATE TABLE 101
Field Definitions 102
Supported Data Types 102
Field Constraints 104
CHECK 104
COMMENT 104
DEFAULT 105
NOT NULL 105
Table Creation Examples 105
Modify Table Definitions 106
ALTER TABLE ADD field 106

ALTER TABLE DROP field 106
DROP TABLE 107
CREATE INDEX 107
DROP INDEX 108
DESCRIBE AS JSON TABLE 108
DESCRIBE AS JSON INDEX 109
SHOW TABLES 109
SHOW INDEXES 109
B. Third Party Licenses 110

Preface

There are two different APIs that can be used to write Oracle NoSQL Database (Oracle NoSQL Database) applications: the original Key/Value API, and the Table API. In addition, the Key/Value API is available in Java and C. The Table API is available in Java, C, node.js (Javascript), and Python. This document describes how to write Oracle NoSQL Database applications using the Table API in Java.

Note

Most application developers should use one of the Table drivers because the Table API offers important features, including secondary indexes. Also, the Key/Value API will eventually be deprecated.

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

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

Conventions Used in This Book

The following typographical conventions are used within in this manual:

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

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

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

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

...

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

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

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

...
```

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

try {
    kvstore = KVStoreFactory.getStore(kconfig);
} catch (FaultException fe) {
    // Some internal error occurred. Either abort your application
    // or retry the operation.
}
```

Note

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

Chapter 1. Developing for Oracle NoSQL Database

You access the data in the Oracle NoSQL Database KVStore using Java drivers that are provided for the product. In addition to the Java drivers, several other drivers are also available. They are:

1. Java Key/Value Driver
2. C Table Driver
3. C Key/Value Driver
4. Python Table Driver
5. node.js Table Driver

Note

New users should use one of the Table drivers unless they require a feature only available in the Key/Value API. The Key/Value API will eventually be deprecated.

The Java and C Key/Value driver provides access to store data using key/value pairs. All other drivers provide access using tables. In addition, the Java Key/Value driver provides Large Object (LOB) support that as of this release does not appear in the other drivers. However, users of the Java Tables driver can access the LOB API, even though the LOB API is accessed using the Key/Value interface.

Users of any of the Table drivers are able to create and use secondary indexing. The Java and C Key/Value drivers do not provide this support.

To work, the C Table, Python Table, and node.js Table drivers require use of a proxy server which translates network activity between the driver and the Oracle NoSQL Database store. The proxy is written in Java, and can run on any machine that is network accessible by both your client code and the Oracle NoSQL Database store. However, for performance and security reasons, Oracle recommends that you run the proxy on the same local host as your driver, and that the proxy be used in a 1:1 configuration with your drivers (that is, each instance of the proxy should be used with just a single driver instance).

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

The rest of this book introduces the Java APIs that you use to access the store, and the concepts that go along with them.

Note

Oracle NoSQL Database is tested with Java 7.

The KVStore Handle

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

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

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

In addition to the `KVStoreConfig` class object, you can also provide a `PasswordCredentials` class object to `KVStoreFactory.getStore()`. You do this if you are using a store that has been configured to require authentication. See Using the Authentication APIs (page 3) for more information.

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

```
package kvstore.basicExample;

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

...

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

The KVStoreConfig Class

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

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

- Consistency

  Consistency is a property that describes how likely it is that a record read from a replica node is identical to the same record stored on a master node. For more information, see Consistency Guarantees (page 83).

- Durability

  Durability is a property that describes how likely it is that a write operation performed on the master node will not be lost if the master node is lost or is shut down abnormally. For more information, see Durability Guarantees (page 90).

- Helper Hosts

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

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

- Request Timeout

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

- Store name

  Identifies the name of the store.

- Password credentials and optionally a reauthentication handler

  See the next section on authentication.

Using the Authentication APIs

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

Authentication involves sending username/password credentials to the store at the time a store handle is acquired.

A store that is configured to support authentication is automatically configured to communicate with clients using SSL in order to ensure privacy of the authentication and other sensitive information. When SSL is used, SSL certificates need to be installed on the machines where your client code runs in order to validate that the store that is being accessed is trustworthy.

Be aware that you can authenticate to the store in several different ways. You can use Kerberos, or you can specify a `LoginCredentials` implementation instance to `KVStoreFactory.getStore()`. (Oracle NoSQL Database provides the `PasswordCredentials` class as a `LoginCredentials` implementation.) If you use Kerberos, you can either use security properties understood by Oracle NoSQL Database to provide necessary Kerberos information, or you can use the Java Authentication and Authorization Service (JAAS) programming framework.

For information on using `LoginCredentials`, see Authentication using LoginCredentials (page 5). For information on using Kerberos, see Authentication using Kerberos (page 9). For information on using JAAS with Kerberos, see Authentication using Kerberos and JAAS (page 11).

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

Configuring SSL

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

Identifying the Trust Store

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

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

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

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

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

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

Setting the SSL Transport Property

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

1. Identify the location of the trust store by using a `Properties` object to set the `oracle.kv.transport` property. You then use

`KVStoreConfig.setSecurityProperties()` to pass the `Properties` object to your `KVStore` handle.

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

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

Authentication using LoginCredentials

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

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

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

```
import java.util.Properties;

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



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

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

```
    kconfig.setSecurityProperties(secProps);

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

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

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

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

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

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

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

KVStore store = null;
try {
    /*
     * storeName, hostName, port are all strings that would come
     * from somewhere else in your application.
     *
     * Notice that we do not pass in any login credentials.
     * All of that information comes from login.txt
     */
    myStoreHandle =
```

```
        KVStoreFactory.getStore(
            new KVStoreConfig(storeName, hostName + ":" + port))
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Renewing Expired Login Credentials

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

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

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

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

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

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

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

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

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

```
package kvstore.basicExample

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

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

        reauthStore.login(cred);
    }
}
```

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

```
import java.util.Properties;

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

import kvstore.basicExample.MyReauthHandler;

KVStore store = null;
try {
    /*
     * storeName, hostName, port, username, and password are all
```

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

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

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

Authentication using Kerberos

You can authenticate to the store by using Kerberos. To do this, you must already have installed Kerberos and obtained the necessary login and service information. See the *Oracle NoSQL Database Security Guide* for details.

The following is a simple example of obtaining a store handle for a secured store, and using Kerberos to authenticate. Information specific to Kerberos, such as the Kerberos user name, is specified using `KVSecurityConstants` that are set as properties to the `KVStoreConfig` instance which is used to create the store handle.

```
import java.util.Properties;

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

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

/* Set the required security properties */
Properties secProps = new Properties();

/* Set the user name */
secProps.setProperty(KVSecurityConstants.AUTH_USERNAME_PROPERTY,
                    "krbuser");

/* Use Kerberos */
secProps.setProperty(KVSecurityConstants.AUTH_EXT_MECH_PROPERTY,
                     "kerberos");

/* Set SSL for the wire level encryption */
secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                     KVSecurityConstants.SSL_TRANSPORT_NAME);

/* Set the location of the public trust file for SSL */
secProps.setProperty
    (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
    "/home/kv/client.trust");

/* Set the service principal associated with the helper host */
final String servicesDesc =
        "localhost:oraclenosql/localhost@EXAMPLE.COM";
secProps.setProperty(
        KVSecurityConstants.AUTH_KRB_SERVICES_PROPERTY,
        servicesDesc);

/*
 * Set the default realm name to permit using a short name for the
 * user principal
 */
secProps.setProperty(KVSecurityConstants.AUTH_KRB_REALM_PROPERTY,
                     "EXAMPLE.COM");

/* Specify the client keytab file location */
secProps.setProperty(KVSecurityConstants.AUTH_KRB_KEYTAB_PROPERTY,
                     "/tmp/krbuser.keytab");

kconfig.setSecurityProperties(secProps);
```

```
store = KVStoreFactory.getStore(kconfig);
```

Authentication using Kerberos and JAAS

You can authenticate to the store by using Kerberos and the Java Authentication and Authorization Service (JAAS) login API. To do this, you must already have installed Kerberos and obtained the necessary login and service information. See the *Oracle NoSQL Database Security Guide* for details.

The following is a simple example of obtaining a store handle for a secured store, and using Kerberos with JAAS to authenticate.

To use JAAS, you create a configuration file that contains required Kerberos configuration information. For example, the following could be placed in the file named `jaas.config`:

```
oraclenosql {
  com.sun.security.auth.module.Krb5LoginModule required
  principal="krbuser"
  useKeyTab="true"
  keyTab="/tmp/krbuser.keytab";
};
```

To identify this file to your application, set the Java property `java.security.auth.login.config` using the `-D` option when you run your application.

Beyond that, you use `KVSecurityConstants` to specify necessary properties, such as the SSL transport. You can also specify necessary Kerberos properties, such as the Kerberos user name, using `KVSecurityConstants`, or you can use the `KerberosCredentials` class to do this.

```
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.Properties;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

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

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

/* Set the required security properties */
```

```
Properties secProps = new Properties();

/* Set SSL for the wire level encryption */
secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                     KVSecurityConstants.SSL_TRANSPORT_NAME);

/* Set the location of the public trust file for SSL */
secProps.setProperty
    (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
    "/home/kv/client.trust");

/* Use Kerberos */
secProps.setProperty(KVSecurityConstants.AUTH_EXT_MECH_PROPERTY,
                     "kerberos");

/* Set Kerberos properties */
final Properties krbProperties = new Properties();

/* Set the service principal associated with the helper host */
final String servicesPpal =
    "localhost:oraclenosql/localhost@EXAMPLE.COM";
krbProperties.setProperty(KVSecurityConstants.AUTH_KRB_SERVICES_PROPERTY,
                          hostName + ":" + servicesPpal);

/* Set default realm name, because the short name
 * for the user principal is used.
 */
krbProperties.setProperty(KVSecurityConstants.AUTH_KRB_REALM_PROPERTY,
                          "EXAMPLE.COM");

/* Specify Kerberos principal */
final KerberosCredentials krbCreds =
    new KerberosCredentials("krbuser", krbProperties);

try {
    /* Get a login context */
    final Subject subj = new Subject();
    final LoginContext lc = new LoginContext("oraclenosql", subj);

    /* Attempt to log in */
    lc.login();

    /* Get the store using the credentials specified in the subject */
    kconfig.setSecurityProperties(secProps);

    store = Subject.doAs(
        subj, new PrivilegedExceptionAction<KVStore>() {
            @Override
            public KVStore run() throws Exception {
```

```
                return KVStoreFactory.getStore(kconfig, krbCreds, null);
            }
        });
} catch (LoginException le) {
    // LoginException handling goes here
} catch (PrivilegedActionException pae) {
    // PrivilegedActionException handling goes here
} catch (Exception e) {
    // General Exception handling goes here
}
```

Unauthorized Access

Clients which must authenticate to a store are granted some amount of access to the store. This could range from a limited set of privileges to full, complete access. The amount of access is defined by the roles and privileges granted to the authenticating user. Therefore, a call to the Oracle NoSQL Database API could fail due to not having the authorization to perform the operation. When this happens, `UnauthorizedException` will be thrown.

See the *Oracle NoSQL Database Security Guide* for information on how to define roles and privileges for users.

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

```
// Open a store handle, and perform authentication as you do
// as described earlier in this section.

...


try {
    // When you attempt some operation (such as a put or delete)
    // to a secure store, you should catch UnauthorizedException
    // in case the user credentials you are using do not have the
    // privileges necessary to perform the operation.
} catch (UnauthorizedException ue) {
    /*
     * When you see this, either abandon the operation entirely,
     * or log out and log back in with credentials that might
     * have the proper permissions for the operation.
     */
    System.out.println("authorization failed!")
    return;
}
```

Chapter 2. Introduction to Oracle KVLite

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

Note

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

See Using the Authentication APIs (page 3) for information on configuring your client code to connect to a secure store. For information on configuring a store to require authentication, see the *Oracle NoSQL Database Security Guide*.

Starting KVLite

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

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

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

For example:

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

Note

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

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

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

or

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

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

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

Stopping and Restarting KVLite

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

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

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

Verifying the Installation

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

- Start another shell and run:

  ```
  jps -m
  ```

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

- Run the `kvclient` test application:

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

  This should write the release to stdout:

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

- Compile and run the example program:

  1. cd KVHOME
  2. Compile the example:

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

  3. Run the example using all default parameters:

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

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

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

kvlite Utility Command Line Parameter Options

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

Note that you can only specify these options the first time KVLite 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 KVLite 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. Introducing Oracle NoSQL Database Tables and Indexes

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

Note

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

Defining Tables

Before an Oracle NoSQL Database client can read or write to a table in the store, the table must be created. There are several ways to do this, but this manual focuses on using Table DDL Statements. These statements can be submitted to the store using the command line interface (CLI), but the recommended approach is to submit them to the store programmatically. Both methods are described in this section.

The DDL language that you use to define tables is described in Table Data Definition Language Overview (page 100) This section provides a brief overview of how to use that language.

As an introductory example, suppose you wanted to use a table named `myTable` with four columns per row: `item`, `description`, `count`, and `percentage`. To create this table, you would use the following statement:

```
CREATE TABLE myTable (
  item STRING,
  description STRING,
  count INTEGER,
  percentage DOUBLE,
  PRIMARY KEY (item) // Every table must have a primary key
)
```

Note

Primary keys are a concept that have not yet been introduced in this manual. See Primary and Shard Key Design (page 27) for a complete explanation on what they are and how you should use them.

To add the table definition to the store, you can add it programmatically using the `KVStore.execute()` or `KVStore.executeSync()` methods. (The latter method executes the statement synchronously.) In order to use these methods, you must establish a connection to the store. This is described in The KVStore Handle (page 2).

For example:

```
package kvstore.basicExample;

import oracle.kv.FaultException;
import oracle.kv.StatementResult;
import oracle.kv.KVStore;
import oracle.kv.table.TableAPI;
...

// kvstore open omitted

TableAPI tableAPI = kvstore.getTableAPI();
ExecutionFuture future = null;
StatementResult result = null;
String statement = null;

public void createTable() {
    TableAPI tableAPI = store.getTableAPI();
    StatementResult result = null;
    String statement = null;

    try {

        /*
         * Add a table to the database.
         * Execute this statement asynchronously.
         */

        statement =
            "CREATE TABLE myTable (" +
            "item STRING," +
            "description STRING," +
            "count INTEGER," +
            "percentage DOUBLE," +
            "PRIMARY KEY (item))"; // Required"
        result = store.executeSync(statement);

        displayResult(result, statement);

    } catch (IllegalArgumentException e) {
        System.out.println("Invalid statement:\n" + e.getMessage());
    } catch (FaultException e) {
        System.out.println
            ("Statement couldn't be executed, please retry: " + e);
    }
}

private void displayResult(StatementResult result, String statement) {
    System.out.println("===========================");
```

```
    if (result.isSuccessful()) {
        System.out.println("Statement was successful:\n\t" +
            statement);
        System.out.println("Results:\n\t" + result.getInfo());
    } else if (result.isCancelled()) {
        System.out.println("Statement was cancelled:\n\t" +
            statement);
    } else {
        /*
         * statement was not successful: may be in error, or may still
         * be in progress.
         */
        if (result.isDone()) {
            System.out.println("Statement failed:\n\t" + statement);
            System.out.println("Problem:\n\t" +
                result.getErrorMessage());
        } else {
            System.out.println("Statement in progress:\n\t" +
                statement);
            System.out.println("Status:\n\t" + result.getInfo());
        }
    }
}
```

Executing DDL Statements using the CLI

You can execute DDL statements using the CLI's `execute` command. This executes DDL statements synchronously. For example:

```
kv-> execute "CREATE TABLE myTable (
> item STRING,
> description STRING,
> count INTEGER,
> percentage DOUBLE,
> PRIMARY KEY (item))"
Statement completed successfully
kv->
```

Supported Table Data Types

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

Supported data types for Oracle NoSQL Database are:

- Array

  An array of values, all of the same type.

- Binary

  Implemented as a byte array with no predetermined fixed size.

- Boolean
- Double
- Enum

  An enumeration, represented as an array of strings.
- Fixed Binary

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

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

  See the following section.
- String

Record Fields

As described in Defining Child Tables (page 22), you can create child tables to hold subordinate information, such as addresses in a contacts database, or vendor contact information for an inventory system. When you do this, you can create an unlimited number of rows in the child table, and you can index the fields in the child table's rows.

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

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

For example:

```
CREATE TABLE myContactsTable (
    uid STRING,
    surname STRING,
```

```
    familiarName STRING,
    homePhone STRING,
    workPhone STRING,
    homeAddress RECORD (street STRING, city STRING, state STRING,
                zip INTEGER CHECK(zip >= 00000 and zip <= 99999)),
    workAddress RECORD (street STRING, city STRING, state STRING,
                zip INTEGER CHECK(zip >= 00000 and zip <= 99999)),
    PRIMARY KEY(uid))
```

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

```
CREATE TABLE myContactsTable (
    uid STRING,
    surname STRING,
    familiarName STRING,
    homePhone STRING,
    workPhone STRING,
    addresses ARRAY(RECORD (street STRING, city STRING, state STRING,
                zip INTEGER CHECK(zip >= 00000 and zip <= 99999))),
    PRIMARY KEY(uid))
```

Defining Child Tables

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

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

To create a child table, you name the table using the format: <*parentTableName*>.<*childTableName*>. For example, suppose you had the trivial table called `myInventory`:

```
CREATE TABLE myInventory (
  itemCategory STRING,
  description STRING,
  PRIMARY KEY (itemCategory)
)
```

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

```
CREATE TABLE myInventory.itemDetails (
    itemSKU STRING,
    itemDescription STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

Note that when you do this, the child table inherits the parent table's primary key. In this trivial case, the child table's primary key is actually two fields: `itemCategory` and `itemSKU`.

This has several ramifications, one of which is that the parent's primary key fields are retrieved when you retrieve the child table. See Retrieve a Child Table (page 41) for more information.

Table Evolution

In the event that you must update your application at some point after it goes into production, there is a good chance that your tables will also have to be updated to either use new fields or remove existing fields that are no longer in use. You do this through the use of the `ALTER TABLE` statement. See Modify Table Definitions (page 106) for details on this statement.

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

Tables can only be evolved if they have already been added to the store.

For example, the following statements evolve the table that was created in the previous section. Note that these would be submitted to the store, one after another, using either the API or the CLI.

```
ALTER TABLE myInventory.itemDetails (ADD salePrice FLOAT)
```

```
ALTER TABLE myInventory.itemDetails (DROP inventoryCount)
```

Creating Indexes

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

Indexes can be created on any field which is an indexable datatype, including primary key fields. See Indexable Field Types (page 26) for information on the types of fields that can be indexed.

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

Note

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

```
CREATE TABLE myInventory.itemDetails (
    itemSKU STRING,
    itemDescription STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

To create an index, use the `CREATE INDEX` statement. See CREATE INDEX (page 107) for details. For example:

```
CREATE INDEX inventoryIdx on myInventory.itemDetails(inventoryCount)
```

Similarly, to remove an index, use the `DROP INDEX` statement. See DROP INDEX (page 108) for details.

```
DROP INDEX inventoryIdx on myInventory.itemDetails
```

Be aware that adding and dropping indexes can take a long time. You might therefore want to run these operations asynchronously using the `KVStore.execute()` method.

```
package kvstore.basicExample;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import oracle.kv.ExecutionFuture;
import oracle.kv.FaultException;
import oracle.kv.StatementResult;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.table.TableAPI;

...
// Store open skipped
...

public void createIndex() {
    TableAPI tableAPI = store.getTableAPI();
    ExecutionFuture future = null;
    StatementResult result = null;
    String statement = null;

    try {

        statement = "CREATE INDEX inventoryIdx on " +
                    "myInventory.itemDetails(inventoryCount)"
        future = store.execute(statement);
        displayResult(future.getLastStatus(), statement);

        /*
         * Limit the amount of time to wait for the
         * operation to finish.
         */
        result = future.get(3, TimeUnit.SECONDS);
        displayResult(result, statement);

    } catch (IllegalArgumentException e) {
```

```
        System.out.println("Invalid statement:\n" + e.getMessage());
    } catch (FaultException e) {
        System.out.println
            ("Statement couldn't be executed, please retry: " + e);
        cleanupOperation(future);
    } catch (ExecutionException e) {
        System.out.println
            ("Problem detected while waiting for a DDL statement: " +
             e.getCause());
        cleanupOperation(future);
    } catch (InterruptedException e) {
        System.out.println
            ("Interrupted while waiting for a DDL statement: " + e);
        cleanupOperation(future);
    } catch (TimeoutException e) {
        System.out.println("Statement execution took too long: " + e);
        cleanupOperation(future);
    }
}

private void cleanupOperation(ExecutionFuture future) {
    if (future == null) {
        /* nothing to do */
        return;
    }

    System.out.println("Statement:");
    System.out.println(future.getStatement());
    System.out.println("has status: ");
    System.out.println(future.getLastStatus());

    if (!future.isDone()) {
        future.cancel(true);
        System.out.println("Statement is cancelled");
    }
}

private void displayResult(StatementResult result, String statement) {
    System.out.println("===========================");
    if (result.isSuccessful()) {
        System.out.println("Statement was successful:\n\t" +
                           statement);
        System.out.println("Results:\n\t" + result.getInfo());
    } else if (result.isCancelled()) {
        System.out.println("Statement was cancelled:\n\t" +
                           statement);
    } else {
        /*
         * statement wasn't successful: may be in error, or may still be
```

```
         * in progress.
         */
        if (result.isDone()) {
            System.out.println("Statement failed:\n\t" + statement);
            System.out.println("Problem:\n\t" + result.getErrorMessage());
        } else {
            System.out.println("Statement in progress:\n\t" + statement);
            System.out.println("Status:\n\t" + result.getInfo());
        }
    }
}
```

Indexable Field Types

Fields can be indexed only if they are declared to be one of the following types. For all complex types (arrays, maps, and records), the field can be indexed if the ultimate target of the index is a scalar datatype. So a complex type that contains a nested complex type (such as an array of records, for example) can be indexed if the index's target is a scalar datatype contained by the embedded record.

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

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

- Maps

  As is the case with Arrays, you can index a map if the map contains scalar types, or if the map contains a record that contains scalar types.

- Records

  Like Arrays and Maps, you can index fields in an embedded record if the field contains scalar data.

See Indexing Non-Scalar Data Types (page 70) for examples of how to index supported non-scalar types.

Chapter 4. Primary and Shard Key Design

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

Primary Keys

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

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

In this case, the table statement you use to define this table is:

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName)
)
```

However, you can use multiple fields for your primary keys. For example:

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName, productType)
)
```

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

We describe how to retrieve multiple rows from your table in Reading Table Rows (page 39). We show how to delete multiple rows at a time in Using multiDelete() (page 36).

Data Type Limitations

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

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

Partial Primary Keys

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

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

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    productClass STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName, productType, productClass)
)
```

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

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

```
productName
productName, productType
```

Shard Keys

Shard keys identify which primary key fields are meaningful in terms of shard storage. That is, rows which contain the same values for all the shard key fields are guaranteed to be stored on the same shard. This matters for some operations that promise atomicity of the results. (See Executing a Sequence of Operations (page 95) for more information.)

For example, suppose you set the following primary keys:

```
PRIMARY KEY (productType, productName, productClass)
```

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

```
PRIMARY KEY (SHARD(productType, productName), productClass)
```

Note that order matters when it comes to shard keys. The keys must be specified in the order that they are defined as primary keys, with no gaps in the key list. In other words, given the above example, it is impossible to set `productType` and `productClass` as shard keys without also specifying `productName` as a shard key.

Row Data

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

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

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

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

Small properties are:

- name
- gender
- address
- phone number

Large properties are:

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

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

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

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

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

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

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

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

```
CREATE TABLE userInfo (
    surname STRING,
    familiarName STRING,
    gender ENUM (male,female),
    street STRING,
    city STRING,
    state STRING,
    zipcode STRING,
    userPhone STRING,
    publickey1 BINARY,
    publickey2 BINARY,
```

```
    PRIMARY KEY (SHARD(surname), familiarName)
)
```

```
CREATE TABLE userInfo.largeProps (
    propType STRING,
    voiceGreeting BINARY,
    imageFile BINARY,
    PRIMARY KEY (propType)
)
```

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

Note

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

Chapter 5. Writing and Deleting Table Rows

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

Write Exceptions

There are many exceptions that you should handle whenever you perform a write operation to the store. Some of the more common exceptions are described here. For simple cases where you use default policies or are not using a secure store, 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 90).

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

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

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

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

Finally, if you are using a secure store that requires authentication, you can receive `AuthenticationFailureException` or `AuthenticationRequiredException` if you do not provide the proper authentication credentials. When using a secure store, you can also see `UnauthorizedException`, which means you are attempting an operation for which the authenticated user does not have the proper permissions.

Writing Rows to a Table in the Store

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

Remember that you can only write data to a table after it has been added to the store. See Introducing Oracle NoSQL Database Tables and Indexes (page 18) for details.

To write a row to a table in the store:

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

   ### Note

   `TableAPI.getTable()` is an expensive call that requires server side access. From a performance point of view, it is a mistake to call this method whenever you need a table handle. Instead, call this method for all relevant tables in the set up section of your code, and then reuse those handles throughout your application.

2. Use the `Table` instance retrieved in the previous step to create a `Row` interface instance. You use the `Table.createRow()` method to do this.

3. Write to each field in the `Row` using `Row.put()`.

4. Write the new row to the store using `TableAPI.put()`.

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

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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

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

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

Writing Rows to a Child Table

To write to a child table, first create the row in the parent table to which the child belongs. You do this by populating the parent row with data. Then you write the child table's row(s). When you do, you must specify the primary key used by the parent table, as well as the primary key used by the child table's rows.

For example, in Defining Child Tables (page 22) we showed how to create a child table. To write data to that table, do this:

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

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

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

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

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

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

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

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

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

Other put Operations

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

- `TableAPI.putIfAbsent()`

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

- `TableAPI.putIfPresent()`

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

- `TableAPI.putIfVersion()`

  This method will put the row only if the value matches the supplied version information. For more information, see Using Row Versions (page 81).

Deleting Rows from the Store

You delete a single row from the store using the `TableAPI.delete()` method. Rows are deleted based on a `PrimaryKey`, which you obtain using the `Table.createPrimaryKey()` method. You can also require a row to match a specified version before it will be deleted. To do this, use the `TableAPI.deleteIfVersion()` method. Versions are described in Using Row Versions (page 81).

When you delete a row, you must handle the same exceptions as occur when you perform any write operation on the store. See Write Exceptions (page 32) for a high-level description of these exceptions.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
```

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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

Using multiDelete()

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

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

For example, suppose you created a table like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:

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

- Row 2:

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

- Row 3:

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

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

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");
```

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

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

Chapter 6. Reading Table Rows

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

1. Retrieve a single row at a time using the `TableAPI.get()` method.
2. Retrieve rows associated with a shard key (which is based on at least part of your primary keys) using either the `TableAPI.multiGet()` or `TableAPI.multiGetIterator()` methods.
3. Retrieve table rows that share a shard key, or an index key, using the `TableAPI.tableIterator()` method.
4. Retrieve and process records from each shard in parallel using a single key as the retrieval criteria. Use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide parallel scans.
5. Retrieve and process records from each shard in parallel using a sequence of keys as the retrieval criteria. Use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide bulk retrievals.

Each of these are described in the following sections.

Read Exceptions

Several 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 83).

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

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

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

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

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

You can also receive a `MetadataNotFoundException`, which indicates that a client's metadata may be out of sync. It extends `FaultException` and can be caught by applications to trigger the need for a refresh of their metadata, and in particular, Table handles obtained via TableAPI.getTable().

Finally, if you are using a secure store that requires authentication, you can receive `AuthenticationFailureException` or `AuthenticationRequiredException` if you do not provide the proper authentication credentials. When using a secure store, you can also see `UnauthorizedException`, which means you are attempting an operation for which the authenticated user does not have the proper permissions.

Retrieving a Single Row

To retrieve a single row from the store:

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

   ### Note

   `TableAPI.getTable()` is an expensive call that requires server side access. From a performance point of view, it is a mistake to call this method whenever you need a table handle. Instead, call this method for all relevant tables in the set up section of your code, and then reuse those handles throughout your application.

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

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

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

For example, in Writing Rows to a Table in the Store (page 32) we showed a trivial example of storing a table row to the store. The following trivial example shows how to retrieve that row.

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...
```

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

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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

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

Retrieve a Child Table

In Writing Rows to a Child Table (page 34) we showed how to populate a child table with data. To retrieve that data, you must specify the primary key used for the parent table row, as well as the primary key for the child table row. For example:

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// We omit retrieval of the parent table because it is not required.
```

```
Table myChildTable = tableH.getTable("myInventory.itemDetails");

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

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

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

For information on how to iterate over nested tables, see Iterating with Nested Tables (page 49).

Using multiGet()

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

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

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

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:

  itemType: Hats
  itemCategory: baseball
  itemClass: longbill

itemColor: red
itemSize: small
price: 12.07
inventoryCount: 127

- Row 2:

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

- Row 3:

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

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

```
package kvstore.basicExample;

...

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

...

// KVStore handle creation is omitted for brevity

...

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

```
// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

List<Row> myRows = null;

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

You can then iterate over the resulting list as follows:

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

Iterating over Table Rows

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

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

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

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

When using this method, you can optionally specify:

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

  This class also allows you to control how many threads are used to perform the store read. By default this method determines the degree of concurrency based on the number of available processors. You can tune this concurrency by explicitly stating how many threads to use for table retrieval. See Parallel Scans (page 56) for more information.

  Finally, you use this class to specify a consistency policy. See Consistency Guarantees (page 83) for more information.

Note

When using `TableAPI.tableIterator()`, it is important to call `TableIterator.close()` when you are done with the iterator to avoid resource leaks. This is especially true for long-running applications, especially if you do not iterate over the entire result set.

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

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:

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

itemSize: small
price: 12.07
inventoryCount: 127

- Row 2:

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

- Row 3:

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

- Row *n*:

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

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

```
package kvstore.basicExample;

...

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

```
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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

Specifying Field Ranges

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

For example, suppose you defined a table like this:

```
CREATE TABLE myTable (
    surname STRING,
    familiarName STRING,
    userID STRING,
    phonenumber STRING,
```

```
    address STRING,
    email STRING,
    dateOfBirth STRING,
    PRIMARY KEY (SHARD(surname, familiarName), userID)
)
```

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

Given this, you could perform operations for all the rows related to users with a surname of `Smith`, but we can limit the result set to just those users with familiar names that fall alphabetically between `Bob` and `Patricia` by specifying a field range.

A `FieldRange` is created using `Table.createFieldRange()`. This method takes just one argument — the name of the primary key for which you want to set the range.

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

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

```
package kvstore.basicExample;

...

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");
```

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

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

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

Iterating with Nested Tables

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

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

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

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

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

```
CREATE TABLE prodTable (
    prodType STRING,
    typeDescription STRING,
    PRIMARY KEY (prodType)
```

```
)
```

```
CREATE TABLE prodTable.prodCategory (
    categoryName STRING,
    categoryDescription STRING,
    PRIMARY KEY (categoryName)
)
```

```
CREATE TABLE prodTable.prodCategory.item (
    itemSKU STRING,
    itemDescription STRING,
    itemPrice FLOAT,
    vendorUID STRING,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

With tables containing data like this:

- Row 1:

  prodType: Hardware
  typeDescription: Equipment, tools and parts

  - Row 1.1:

    categoryName: Bolts
    categoryDescription: Metric & US Sizes

    - Row 1.1.1:

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

- Row 2:

  prodType: Tools
  typeDescription: Hand and power tools

  - Row 2.1:

    categoryName: Handtools
    categoryDescription: Hammers, screwdrivers, saws

    - Row 2.1.1:

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

inventoryCount: 249

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

Start by getting all our table handles:

```
package kvstore.tableExample;

import java.util.Arrays;

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

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

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

...

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

private static TableAPI tableH;

...

// KVStore handle creation is omitted for brevity

...

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

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

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

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

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

Now we perform the iteration:

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

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

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

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

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

private static void displayCategoryTableRow(Row row) {
    System.out.println("\tCategory: " +
```

```
            row.get("categoryName").asString().get());
        System.out.println("\tDescription: " +
            row.get("categoryDescription").asString().get());
}

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

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

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

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

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

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

```
        iter.close();
    }
}
```

Reading Indexes

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

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

For example, suppose you defined a table like this:

```
CREATE TABLE myTable (
    surname STRING,
    familiarName STRING,
    userID STRING,
    phonenumber STRING,
    address STRING,
    email STRING,
    dateOfBirth STRING,
    PRIMARY KEY (SHARD(surname, familiarName), userID)
)
```

```
CREATE INDEX DoB ON myTable (dateOfBirth)
```

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

```
package kvstore.basicExample;

...

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

...

// KVStore handle creation is omitted for brevity

...
```

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

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

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

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

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

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

```
package kvstore.basicExample;

...

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

...
```

```
// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

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

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

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

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

Parallel Scans

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

Note

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

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

```
package kvstore.basicExample;

...

import oracle.kv.Consistency;
import oracle.kv.Direction;
import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.MultiRowOption;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.TableIteratorOptions;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

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

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

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

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

Bulk Get Operations

Bulk get operations allow you to retrieve and process records from each shard in parallel, like a parallel scan, but using a sequence of keys instead of a single key as retrieval criteria.

A bulk get operation does not return the entire set of rows all at once. Instead, it batches the fetching of rows in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth. Batches are fetched in parallel across multiple Replication Nodes. If more threads are specified on the client side, then the user can expect better retrieval performance – until processor or network resources are saturated.

To do this, use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide bulk retrievals. These accept a sequence of keys instead of a single key as the retrieval criteria. The sequence is provided using either an `Iterator<Key>` or `List<Iterator<Key>>` value.

The methods retrieve the rows or primary keys matching the keys supplied by the iterator(s).

Note

If the iterator yields duplicate keys, the row associated with the duplicate keys will be returned at least once and potentially multiple times.

The supplied keys should follow these rules:

1. All supplied primary keys should belong to the same table.
2. The input key must be a complete shard key.
3. If a field range is specified, then the partial primary keys should be uniform. That is, they should have the same number of components. Also, the field range must be the first unspecified field of the supplied key.

When using these methods, you can also optionally specify:

- A `MultiRowOptions` class instance which allows you to specify a field range, as well as the ancestor and parent tables you want to include in the iteration.
- The number of keys to fetch during each network round trip using a `TableIteratorOptions` class instance. If you provide a value of 0, an internally determined default is used. You can also specify the traversal order (`UNORDERED` is supported).

You can control how many threads are used to perform the store read using the `MaxConcurrentRequests` parameter.

Finally, you can specify a consistency policy. See Consistency Guarantees (page 83) for more information.

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

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory), itemClass, itemColor,
    itemSize))
```

With tables containing data like this:

- Row 1:

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

- Row 2:

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

- Row 3:

  itemType: Pants
  itemCategory: baseball
  itemClass: Summer
  itemColor: red
  itemSize: large
  price: 14.07

inventoryCount: 39

- Row 4:

  itemType: Pants
  itemCategory: baseball
  itemClass: Winter
  itemColor: white
  itemSize: large
  price: 16.99
  inventoryCount: 9

- Row *n*:

  itemType: Coats
  itemCategory: Casual
  itemClass: Winter
  itemColor: red
  itemSize: large
  price: 247.99
  inventoryCount: 13

If you want to locate all the Hats and Pants used for baseball, using nine threads in parallel, you can retrieve all of the records as follows:

```
package kvstore.basicExample;

...
import java.util.ArrayList;
import java.util.List;
import oracle.kv.Consistency;
import oracle.kv.Direction;
import oracle.kv.table.MultiRowOptions;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.TableIteratorOptions;

...

// KVStore handle creation is omitted for brevity

...

// Construct the Table Handle
TableAPI tableH = store.getTableAPI();
Table table = tableH.getTable("myTable");


// Use multi-threading for this store iteration and limit the number
```

```
// of threads (degree of parallelism) to 9.
final int maxConcurrentRequests = 9;
final int batchResultsSize = 0;
final TableIteratorOptions tio =
   new TableIteratorOptions(Direction.UNORDERED,
                  Consistency.NONE_REQUIRED,
                  0, null,
                  maxConcurrentRequests,
                  batchResultsSize);

// Create retrieval keys
PrimaryKey myKey = table.createPrimaryKey();
myKey.put("itemType", "Hats");
myKey.put("itemCategory", "baseball");
PrimaryKey otherKey = table.createPrimaryKey();
otherKey.put("itemType", "Pants");
otherKey.put("itemCategory", "baseball");

List<PrimaryKey> searchKeys = new ArrayList<PrimaryKey>();

// Add the retrieval keys to the list.
searchKeys.add(myKey);
searchKeys.add(otherKey);


final TableIterator<Row> iterator = tableH.tableIterator(
                                     searchKeys.iterator(), null, tio);

// Now retrieve the records.
try {
    while (iterator.hasNext()) {
    Row row = (Row) iterator.next();
    // Do some work with the Row here
    }
} finally {
   if (iterator != null) {
   iterator.close();
   }
}
```

Chapter 7. Using Data Types

Many of the types that Oracle NoSQL Database offers are easy to use. Examples of their usage has been scattered throughout this manual. However, some types are a little more complicated to use because they use container methods. This chapter describes their usage.

The types described in this chapter are: Arrays, Maps, Records, Enums, and Byte Arrays. This chapter shows how to read and write values of each of these types.

Using Arrays

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

When you declare a table field as an array, you use the `ARRAY()` statement.

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

```
CREATE TABLE myTable (
    uid INTEGER,
    myArray ARRAY(STRING),
    PRIMARY KEY(uid)
)
```

`CHECK` constraints are supported for array values. See CHECK (page 104) for more details.

`DEFAULT` and `NOT NULL` constraints are not supported for arrays.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using Binary

You can declare a field as binary using the `BINARY` statement. You then read and write the field value using a Java byte array.

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

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

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

```
CREATE TABLE myTable (
    uid INTEGER,
```

```
    myByteArray BINARY,
    PRIMARY KEY(uid)
)
```

`CHECK`, `DEFAULT` and `NOT NULL` constraints are not supported for binary values.

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

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

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

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

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

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

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

For example:

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

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

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

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

Using Enums

Enumerated types are declared using the `ENUM()` statement. You must declare the acceptable enumeration values when you use this statement.

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

```
CREATE TABLE myTable (
    uid INTEGER,
```

```
    myEnum ENUM (Apple,Pears,Oranges),
    PRIMARY KEY (uid)
)
```

CHECK constraints are not supported for enumerated fields.

DEFAULT and NOT NULL constraints are supported for enumerated fields. See DEFAULT (page 105) for more information.

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

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

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

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

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

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

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

For example:

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

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

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

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

Using Fixed Binary

You can declare a fixed binary field using the BINARY() statement. When you do this, you must also specify the field's size in bytes. You then read and write the field value using Java byte arrays. However, if the byte array does not equal the specified size, then

`IllegalArgumentException` is thrown when you attempt to write the field. Write the field value using a Java byte array.

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

Fixed binary should be used over the binary datatype any time you know that all the field values will be of the same size. Fixed binary is a more compact storage format because it does not need to store the size of the array. See Using Binary (page 63) for information on the binary datatype.

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

```
CREATE TABLE myTable (
    uid INTEGER,
    myByteArray BINARY(20),
    PRIMARY KEY (uid)
)
```

`CHECK`, `DEFAULT` and `NOT NULL` constraints are not supported for binary values.

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

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

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

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

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

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

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

For example:

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

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

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

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

Using Maps

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

The string "[]" is reserved and must not be used for key names.

When you declare a table field as a map, you use the `MAP()` statement. You must also declare the map element's data types.

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

```
CREATE TABLE myTable (
    uid INTEGER,
    myMap MAP(INTEGER),
    PRIMARY KEY (uid)
)
```

`CHECK` constraints are supported for map fields. See CHECK (page 104) for more information.

`DEFAULT` and `NOT NULL` constraints are not supported for map fields.

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

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

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

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

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

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

To read the map, use `Row.get().asMap()`. This returns a `MapValue` class instance. You can then use `MapValue.get()` to retrieve an map value. The retrieved value is returned as a

FieldValue, which allows you to retrieve the encapsulated value using a cast method such as FieldValue.asInteger().

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

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

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

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

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

Using Embedded Records

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

Use the RECORD() statement to declare a table field as a record.

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

```
CREATE TABLE myTable (
    uid INTEGER,
    myRecord RECORD(firstField STRING, secondField INTEGER),
    PRIMARY KEY (uid)
)
```

CHECK, DEFAULT and NOT NULL constraints are not supported for embedded record fields. However, these constraints can be applied to the individual fields in an embedded record. See Field Constraints (page 104) for more information.

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

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

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

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

RecordValue rv = row.putRecord("myRecord");
rv.put("firstField", "An embedded record STRING field");
```

```
rv.put("secondField", 3388);

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

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

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

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

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

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

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

Chapter 8. Indexing Non-Scalar Data Types

We describe how to index scalar data types in Creating Indexes (page 23), and we show how to read using indexes in Reading Indexes (page 54). However, non-scalar data types (Arrays, Maps and Records) require more explanation, which we give here.

Index creation is accomplished using the `CREATE INDEX` statement. See CREATE INDEX (page 107) for details on this statement.

Indexing Arrays

You can create an index on an array field so long as the array contains scalar data, or contains a record with scalar fields.

Note

You cannot index a map or array that is nested beneath another map or array. This is not allowed because of the potential for an excessively large number of index entries.

Be aware that indexing an array potentially results in multiple index entries for each row, which can lead to very large indexes.

To create the index, first create the table:

```
CREATE TABLE myArrayTable (
    uid INTEGER,
    testArray ARRAY(STRING),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index:

```
CREATE INDEX arrayFieldIndex on myArrayTable (testArray)
```

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

To retrieve data using an index of arrays, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index arrayIndex = myTable.getIndex("arrayFieldIndex");
IndexKey indexKey = arrayIndex.createIndexKey();
```

Next you create an `ArrayValue` instance, and assign a value to it.

When you perform the index lookup, the only records that will be returned will be those which have an array with at least one item matching the value set for the `ArrayValue`. For example, if you have individual records that contain arrays like this:

```
Record 1: ["One," "Two", "Three"]
```

```
Record 2: ["Two", "Three", "One"]
Record 3: ["One", "Three", "One"]
Record 4: ["Two", "Three", "Four"]
```

and you then perform an array lookup on the array value "One", then Records 1 - 3 will be returned, but not 4.

To set an `ArrayValue`, construct it using `IndexKey.putArray()`. Pass the method the name of the array field. Then add the index value to the `ArrayValue` instance using an array of the appropriate type:

```
ArrayValue av = indexKey.putArray("testArray");
av.add("One");
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for Array value 'One' : ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        ArrayValue avRet = rowRet.get("testArray").asArray();
        for (FieldValue fv: avRet.toList()) {
            System.out.println(fv.asString().get());
        }
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing Maps

You can create an index on a map field so long as the map contains scalar data, or contains a record with scalar fields.

Note

You cannot index a map or array that is nested beneath another map or array. This is not allowed because of the potential for an excessively large number of index entries.

To create the index, define the map as normal. Once the map is defined for the table, there are several different ways to index it:

- Based on the map's keys without regard to the actual key values.
- Based on the map's values, without regard to the actual key used.

- By a specific map key. To do this, you specify the name of the map field *and* the name of a map key using dot notation. If the map key is ever created using your client code, then it will be indexed.
- Based on the map's key and value without identifying a specific value (such as is required by the previous option in this list).

Indexing by Map Keys

You can create indexes based on a map's keys without regard to the corresponding values.

Be aware that creating an index like this can potentially result in multiple index entries for each row, which can lead to very large indexes. In addition, the same row can appear in a result set, so the duplicate entries must be handled by your application.

First create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index using the `KEYOF` statement:

```
CREATE INDEX mapKeyIndex on myMapTable (KEYOF(testMap))
```

Data is retrieved if the table row contains the identified map with the identified key. So, for example, if you create a series of table rows like this:

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

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

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
```

```
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with any key currently in use by the map. For example, "field3".

To retrieve data using a map index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapKeyIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next you create and populate a `MapValue` instance, using the `MapValue.putNull()` method. This method allows you to retrieve the map entry without regard to the value part of the entry. When you perform the index lookup, the only records that will be returned will be those which have a map with the specified key name:

```
MapValue mv = indexKey.putMap("testMap");
mv.putNull("field3");
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing by Map Values

You can create indexes based on the values contained in a map without regard to the keys in use.

Be aware that creating an index like this can potentially result in multiple index entries for each row, which can lead to very large indexes. In addition, the same row can appear in a result set, so the duplicate entries must be handled by your application.

First create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
```

```
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index using the `ELEMENTOF` statement:

```
CREATE INDEX mapElementIndex on myMapTable (ELEMENTOF(testMap))
```

Data is retrieved if the table row contains the identified map with the identified value. So, for example, if you create a series of table rows like this:

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

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

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with any value currently in use by the map. For example, a value of "2".

To retrieve data using a map index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapElementIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next you create and populate a `MapValue` instance, using the `MapValue.put()` method. For the `key` value for this method, use `MapValue.ANONYMOUS`. For the `value` field, specify `2`. When you perform the index lookup, the only records that will be returned will be those which have a map with a value of `2`.

```
MapValue mv = indexKey.putMap("testMap");
```

```
mv.put(MapValue.ANONYMOUS, 2);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap value 2: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing by a Specific Map Key Name

You can create an index based on a specified map key name. Any map entries containing the specified key name are indexed. This can create a small and very efficient index because the index does not contain every key/value pair contained by the map fields. Instead, it just contains those map entries using the identified key, which results in at most a single index entry per row.

To create the index, first create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index by specifying the key name you want indexed using dot notation. In this example, we will index the key name of "field3":

```
CREATE INDEX mapField3Index on myMapTable (testMap.field3)
```

Data is retrieved if the table row contains the identified map with the indexed key and a specified value. So, for example, if you create a series of table rows like this:

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

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

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
```

```
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with key "field3" (because that is what you indexed) when "field3" maps to a specified value — such as "3". If you try to do an index lookup on, for example, "field2" then that will fail because you did not index "field2".

To retrieve data using a map index, you first retrieve the index using its name and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapField3Index");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next you create and populate a `MapValue` instance using `MapValue.put()`. When you perform the index lookup, the only records that will be returned will be those which have a map with the matching key name and corresponding value.

```
MapValue mv = indexKey.putMap("testMap");
mv.put("field3", 3);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3, value 3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
```

```
    }
}
```

Indexing by Map Key and Value

In the previous section, we showed how to create a map index by specifying a pre-determined key name. This allows you to perform map index look ups by providing both key and value, but the index lookup will only be successful if the specified key is the key that you indexed.

You can do the same thing in a generic way by indexing every key/value pair in your map. The result is a more flexible index, but also an index that is potentially much larger than the previously described method. It is likely to result in multiple index entries per row.

To create an index based on every key/value pair used by the map field, first create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index by using both the `KEYOF` and `ELEMENTOF` keywords:

```
CREATE INDEX mapKeyValueIndex on myMapTable \
(KEYOF(testMap),ELEMENTOF(testmap))
```

Data is retrieved if the table row contains the identified map with the identified key and the identified value. So, for example, if you create a series of table rows like this:

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

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

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
```

```
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with specified key/value pairs — for example, key "field3" and value "3".

To retrieve data using a map index, you first retrieve the index using its name and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapKeyValueIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next you create and populate a `MapValue` instance using `MapValue.put()`. When you perform the index lookup, the only records that will be returned will be those which have a map with the matching key/value pair.

```
MapValue mv = indexKey.putMap("testMap");
mv.putNull("field3");
mv.put(MapValue.ANONYMOUS, 3);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3, value 3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing Embedded Records

You can create an index on an embedded record field so long as the record field contains scalar data. To create the index, define the record as normal. To index the field, you specify the name of the embedded record *and* the name of the field using dot notation.

To create the index, first create the table:

```
CREATE Table myRecordTable (
    uid INTEGER,
    myRecord RECORD (firstField STRING, secondField INTEGER),
    PRIMARY KEY (uid)
```

```
)
```

Once the table has been added to the store, create the index:

```
CREATE INDEX recordFieldIndex on myRecordTable (myRecord.secondField)
```

Data is retrieved if the table row contains the identified record field with the specified value. So, for example, if you create a series of table rows like this:

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

Row row = myTable.createRow();
row.put("uid", 12345);
RecordValue rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 12345");
rv.put("secondField", 3388);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 345);
rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 345");
rv.put("secondField", 3388);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 111);
rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 111");
rv.put("secondField", 12);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the embedded record where "secondField" is set to a specified value. (The embedded record index that we specified, above, indexed myRecord.secondField.)

To retrieve data using a record index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index recordIndex = myTable.getIndex("recordFieldIndex");
IndexKey indexKey = recordIndex.createIndexKey();
```

Next you create a `RecordValue` instance, and assign a value to it. When you perform the index lookup, the only records that will be returned will be those which have an embedded record with the specified field and field value.

To set a `RecordValue`, construct it using `IndexKey.putRecord()`. Pass the method the name of the record field. Then add the index value to the `RecordValue` instance:

```
RecordValue rv = indexKey.putRecord("myRecord");
rv.put("secondField", 3388);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testRecord.secondField, value 3388: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        RecordValue recordRet = rowRet.get("myRecord").asRecord();
        System.out.println("myRecord: " + recordRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Chapter 9. Using Row Versions

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

There are two reasons why versions might be important.

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

2. When a client reads data that was previously written, it may be important to ensure that the Oracle NoSQL Database node servicing the read operation has been updated with the information previously written. This can be accomplished by passing the version of the previously written data as a consistency parameter to the read operation. For more information on using consistency, see Consistency Guarantees (page 83).

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

The following code fragment retrieves a row, and then writes that row back to the store only if the version has not changed:

```
package kvstore.basicExample;

...

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

...

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

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

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

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

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

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

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

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

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

Chapter 10. Consistency Guarantees

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

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

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

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

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

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

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

Specifying Consistency Policies

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

Once you have selected a consistency policy, you can put it to use in one of two ways. First, you can use it to define a default consistency policy using the `KVStoreConfig.setConsistency()` method. Specifying a consistency policy in this way 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 a `ReadOption` class instance you provide to the `TableAPI` method that you are using to perform the store read operation.

The following example shows how to set a default consistency policy for the store. We will show the per-operation method of specifying consistency policies 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 Simple Consistency

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

1. `Consistency.ABSOLUTE`

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

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

2. `Consistency.NONE_REQUIRED`

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

3. `Consistency.NONE_REQUIRED_NO_MASTER`

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

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

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

```
package kvstore.basicExample;

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

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

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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


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

Using Time-Based Consistency

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

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

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

- `permissibleLag`

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

- `permissibleLagUnits`

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

- `timeout`

  A `long` that describes how long the replica is permitted to wait in an attempt to meet the permissible lag limit. That is, if the replica cannot immediately meet the permissible lag 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 permissible lag 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, use the `Consistency.Version` class. When you do this, you must provide the following information:

- `version`

  The `Version` that the read must match.

- `timeout`

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

- `timeoutUnit`

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

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

```
package kvstore.basicExample;

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

...

// KVStore handle creation is omitted for brevity

...

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

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

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

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

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

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

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

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

```
package kvstore.basicExample;

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

import java.util.concurrent.TimeUnit;

...

// KVStore handle creation is omitted for brevity

...

// Construct the PrimaryKey.

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

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

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

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

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

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

Chapter 11. Durability Guarantees

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

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

Note

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

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

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

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

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

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

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

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

Setting Acknowledgment-Based Durability Policies

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

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

Note

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

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

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

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

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

Setting Synchronization-Based Durability Policies

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

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

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

- NO_SYNC

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

- WRITE_NO_SYNC

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

- SYNC

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

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

Setting Durability Guarantees

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

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

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

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

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

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

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

- All replicas must acknowledge the write operation.

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

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

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

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

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

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

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

```
package kvstore.basicExample;

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

...

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

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

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

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

```
package kvstore.basicExample;

...
```

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

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

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

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

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

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

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

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

Chapter 12. Executing a Sequence of Operations

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

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

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

When using a sequence of operations:

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

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

Sequence Errors

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

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

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

Creating a Sequence

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

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

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:

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

- Row 2:

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

- Row 3:

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

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

Then you can create a sequence in the following way:

```
package kvstore.basicExample;
```

```
import java.util.ArrayList;

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

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

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

...

// kvstore handle creation omitted.

...

TableAPI tableH = kvstore.getTableAPI();

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

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

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

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

row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
```

```
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "medium");
row.put("price", new Float(14.07));
row.put("inventoryCount", 198);
opList.add(tof.createPut(row, null, true));

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

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

Executing a Sequence

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

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

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

    // Or more than one operation uses the same key.
} catch (DurabilityException de) {
    // The durability guarantee could not be met.
} catch (RequestTimeoutException rte) {
    // The operation was not completed inside of the
```

```
    // default request timeout limit.
} catch (FaultException fe) {
    // A generic error occurred
}
```

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

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

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

For an example of using `WriteOptions`, see Durability Guarantees (page 90).

Appendix A. Table Data Definition Language Overview

Before you can write data to tables in the store, you must provide a definition of the tables you want to use. This definition includes information such as the table's name, the name of its various rows and the data type contained in those rows, identification of the primary and (optional) shard keys, and so forth. To perform these definitions, Oracle NoSQL Database provides a Data Definition Language (DDL) that you use to form table and index statements. These statements can be used to:

- Define tables and sub-tables.
- Modify table definitions.
- Delete table definitions.
- Define indexes.
- Delete index definitions.

Table and index statements take the form of ordinary strings, which are then transmitted to the Oracle NoSQL Database store using the appropriate method or function. For example, to define a simple user table, the table statement might look like this:

```
CREATE TABLE Users (
    id INTEGER,
    firstName STRING,
    lastName STRING,
    PRIMARY KEY (id)
)
```

For information on how to transmit these statements to the store, see Introducing Oracle NoSQL Database Tables and Indexes (page 18).

For overview information on primary and shard keys, see Primary and Shard Key Design (page 27).

For overview information on indexes, see Creating Indexes (page 23).

The remainder of this appendix describes in detail the DDL statements that you use to manipulate table and index definitions in the store.

Name Constraints

The following sections use all uppercase to identify DDL keywords (such as STRING, CHECK, CREATE TABLE, and so on). However, these keywords are actually case-insensitive and can be entered as lower-case.

The DDL keywords shown here are reserved and cannot be used as table, index or field names.

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

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

DDL Comments

You can include comments in your DDL statements using one of the following constructs:

```
id INTEGER, /* this is a comment */
firstName STRING, // this is a comment
lastName STRING, # this is a comment
```

CREATE TABLE

To create a table definition, use a `CREATE TABLE` statement. Its form is:

```
CREATE TABLE [IF NOT EXISTS] table-name (
    field-definition, field-definition-2 ...,
    PRIMARY KEY (field-name, field-name-2...),
    [COMMENT "comment string"]
)
```

where:

- `IF NOT EXISTS` is optional, and it causes table creation to be silently skipped if a table of the given name already exists in the store, and the table's definition exactly matches the provided definition. No error is returned as a result of the statement execution.

  If this statement phrase is not specified, then an attempt to duplicate a table name in the store results in a failed table creation.

- *table-name* is the name of the table. This field is required. If you are creating a sub-table, then use dot notation. For example, a table might be named `Users`. You might then define a sub-table named `Users.MailingAddress`.

- *field-definition* is a comma-separated list of fields. There are one or more field definitions for every table. Field definitions are described next in this section.

- `PRIMARY KEY` identifies at least one field in the table as the primary key. A primary key definition is required for every table. For information on primary keys, see Primary Keys (page 27).

  To define a shard key (optional), use the SHARD keyword in the primary key statement. For information on shard keys, see Shard Keys (page 28).

  For example:

  ```
  PRIMARY KEY (SHARD(id), lastName)
  ```

- `COMMENT` is optional. You can use this to provide a brief description of the table. The comment will not be interpreted but it is included in the table's metadata.

Field Definitions

When defining a table, field definitions take the form:

```
field-name type [constraints] [COMMENT "comment-string"]
```

where:

- *field-name* is the name of the field. For example: `id` or `familiarName`. Every field must have a name.

- *type* describes the field's data type. This can be a simple type such as INTEGER or STRING, or it can be a complex type such a RECORD. The list of allowable types is described in the next section.

- *constraints* describes any limits placed on the data contained in the field. That is, minimum or maximum values, allowable ranges, or default values. This information is optional. See Field Constraints (page 104) for more information.

- `COMMENT` is optional. You can use this to provide a brief description of the field. The comment will not be interpreted but it is included in the table's metadata.

Supported Data Types

The following data types are supported for table fields:

- `ARRAY`

  An array of data. All elements of the array must be of the same data type, and this type must be declared when you define the array field. For example, to define an array of strings:

  ```
  myArray ARRAY(STRING)
  ```

  Note that field constraints can be applied to array value. For example:

  ```
  myArray ARRAY(INTEGER CHECK(ELEMENTOF(myArray) > 0 and \
  ELEMENTOF(myArray) < 100))
  ```

  See CHECK (page 104) for a description of the CHECK statement.

- `BINARY`

  Binary data.

- `BINARY(length)`

  Fixed-length binary field of size *length* (in bytes).

- `BOOLEAN`

A boolean data type.

- `DOUBLE`

  A double.

- `ENUM`

  An enumerated list. The field definition must provide the list of allowable enumerated values. For example:

  ```
  fruitName ENUM(apple,pear,orange)
  ```

- `FLOAT`

  A float.

- `INTEGER`

  An integer.

- `LONG`

  A long.

- `MAP`

  A data map. All map keys are strings, but when defining these fields you must define the data type of the data portion of the map. For example, if your keys map to integer values, then you define the field like this:

  ```
  myMap MAP(INTEGER)
  ```

  Note that field constraints can be applied to mapped value. For example:

  ```
  myMap MAP(INTEGER CHECK(ELEMENTOF(myMap) > 0 and \
  ELEMENTOF(myMap) < 13))
  ```

  See CHECK (page 104) for a description of the CHECK statement.

- `RECORD`

  An embedded record. This field definition must define all the fields contained in the embedded record. All of the same syntax rules apply as are used for defining an ordinary table field. For example, a simple embedded record might be defined as:

  ```
  myEmbeddedRecord RECORD(firstField STRING, secondField INTEGER)
  ```

  Data constraints, default values, and so forth can also be used with the embedded record's field definitions.

- `STRING`

  A string.

Field Constraints

Field constraints are used to define information about the field, such as the allowable range of values and default values. For example:

```
day_of_month CHECK (day_of_month >= 1 AND day_of_month <= 31)
```

Not all data type support constraints, and individual data types do not support all possible constraints.

CHECK

Use CHECK to specify an allowable range of values. The symbols AND, <, <=, >, and >= are all supported. <= and >= specifying inclusive ranges, and < and > specify exclusive ranges. For example:

```
myInt INTEGER CHECK(myInt > 10 and myInt < 20)
```

For simple data types, (INTEGER, LONG, FLOAT, DOUBLE, STRING), use the field's name to specify the range, as shown in the previous example.

For STRING datatypes, the range specifies the string's value range based on a lexicographical comparison of the Unicode value of each character in the string. For example:

```
 myString STRING CHECK(myString > "aaa" and myString < "zzz")
```

causes the string ccc to be within the valid range, but CCC or cccc would not be. If you specify numbers for the range, then the number is interpreted as a string range. In this case:

```
 myString STRING CHECK(myString > 10 and myString < 20)
```

means that 11 is allowable, but 21 or aaa would not be.

For MAP and ARRAY datatypes, CHECK can be used to constraint the range of allowable values. Use ELEMENTOF() to refer to the MAP's or ARRAY's value. For example:

```
myMap MAP(INTEGER CHECK(ELEMENTOF(myMap) > 10))
```

or:

```
myArray ARRAY(INTEGER CHECK(ELEMENTOF(myArray) > 100 AND \
ELEMENTOF(myArray) < 1000))
```

CHECK is not supported for BINARY, BOOLEAN, ENUM, or RECORD datatypes, although CHECK is supported for the individual fields defined by RECORD:

```
myRec RECORD(a STRING, b INTEGER CHECK(b >= 0 AND b <= 10))
```

COMMENT

All data types can accept a COMMENT as part of their constraint. COMMENT strings are not parsed, but do become part of the table's metadata. For example:

```
myRec RECORD(a STRING, b INTEGER) COMMENT "Comment string"
```

or

```
myInt INTEGER CHECK(myInt > 10 and myInt < 20) COMMENT "Comment string"
```

DEFAULT

All fields can accept a DEFAULT constraint, except for ARRAY, BINARY, MAP, and RECORD. The value specified by DEFAULT is used in the event that the field data is not specified when the table is written to the store.

For example:

```
id INTEGER DEFAULT -1,
description STRING DEFAULT "NONE",
size ENUM(small,medium,large) DEFAULT medium,
inStock BOOLEAN DEFAULT FALSE
```

NOT NULL

`NOT NULL` indicates that the field cannot be NULL. This constraint requires that you also specify a DEFAULT value. Order is unimportant for these constraints. For example:

```
id INTEGER NOT NULL DEFAULT -1,
description STRING DEFAULT "NONE" NOT NULL
```

Table Creation Examples

The following are provided to illustrate the concepts described above.

```
CREATE TABLE users (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  PRIMARY KEY (id),
  COMMENT "This comment applies to the table itself"
)
```

```
CREATE TABLE usersNoId (
  firstName STRING,
  lastName STRING COMMENT "This comment applies to this field only",
  age INTEGER CHECK (age > 0 AND age < 150),
  ssn STRING NOT NULL DEFAULT "xxx-yy-zzzz",
  PRIMARY KEY (SHARD(lastName), firstName)
)
```

```
CREATE TABLE users.address (
  streetNumber INTEGER,
  streetName STRING,  // this comment is ignored by the DDL parser
  city STRING,
  /* this comment is ignored */
  zip INTEGER CHECK(zip > 11111 AND zip < 99999),
  addrType ENUM (home, work, other),
  PRIMARY KEY (addrType)
)
```

```
CREATE TABLE complex (
  COMMENT "this comment goes into the table metadata"
  id INTEGER,
  PRIMARY KEY (id), # this comment is just syntax
  nestedMap MAP(RECORD( m MAP(FLOAT), a ARRAY(RECORD(age INTEGER)))),
  address RECORD (street INTEGER, streetName STRING, city STRING, \
                  zip INTEGER COMMENT "zip comment"),
  friends MAP (STRING),
  floatArray ARRAY (FLOAT),
  aFixedBinary BINARY(5),
  days ENUM(mon, tue, wed, thur, fri, sat, sun) NOT NULL DEFAULT tue
)
```

Modify Table Definitions

Use `ALTER TABLE` statements to either add new fields to a table definition, or delete a currently existing field definition.

You cannot modify an existing field directly. Instead, you must delete the field, then add the field back using the new definition. Note that this will cause all existing data associated with the current field to be deleted.

ALTER TABLE ADD field

To add a field to an existing table, use the `ADD` statement:

```
ALTER TABLE table-name (ADD field-definition)
```

See Field Definitions (page 102) for a description of what should appear in *field-definitions*, above. For example:

```
ALTER TABLE Users (ADD age INTEGER)
```

You can also add fields to nested records. For example, if you have the following table definition:

```
CREATE TABLE u (id INTEGER,
                info record(firstName String)),
                PRIMARY KEY(id))
```

then you can add a field to the nested record by using dot notation to identify the nested table, like this:

```
ALTER TABLE u(ADD info.lastName STRING)
```

ALTER TABLE DROP field

To delete a field from an existing table, use the `DROP` statement:

```
ALTER TABLE table-name (DROP field-name)
```

For example, to drop the `age` field from the `Users` table:

```
ALTER TABLE Users (DROP age)
```

Note that you cannot drop a field if it is the primary key.

DROP TABLE

To delete a table definition, use a `DROP TABLE` statement. Its form is:

```
DROP TABLE [IF EXISTS] table-name
```

where:

- `IF EXISTS` is optional, and it causes the drop statement to be ignored if a table with the specified name does not exist in the store. If this phrase is not specified, and the table does not currently exist, then the DROP statement will fail with an error.

- *table-name* is the name of the table you want to drop.

Note that dropping a table is a lengthy operation because all table data currently existing in the store is deleted as a part of the drop operation.

If child tables are defined for the table that you are dropping, then they must be dropped first. For example, if you have tables:

myTable
myTable.childTable1
myTable.childTable2

then `myTable.childTable1` and `myTable.childTable2` must be dropped before you can drop `myTable`.

CREATE INDEX

To add an index definition to the store, use a `CREATE INDEX` statement. Its form is:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (field-name)
```

When indexing a map field, the previous syntax is acceptible, as are any of the following:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (KEYOF(field-name))
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(ELEMENTOF(field-name))
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(KEYOF(field-name),ELEMENTOF(field-name))
```

where:

- `IF NOT EXISTS` is optional, and it causes the `CREATE INDEX` statement to be ignored if an index by that name currently exists. If this phrase is not specified, and an index using the specified name does currently exist, then the `CREATE INDEX` statement will fail with an error.

- *index-name* is the name of the index you want to create.
- *table-name* is the name of the table that you want to index.
- *field-name* is the name of the field that you want to index.
- `KEYOF` is a keyword that causes index entries to be created based on keys contained in a map.
- `ELEMENTOF` is a keyword that causes index entries to be created based on the values contained in a map.

For example, if table `Users` has a field called `lastName`, then you can index that field with the following statement:

```
CREATE INDEX surnameIndex ON Users (lastName)
```

Note that depending on the amount of data in your store, creating indexes can take a long time. This is because index creation requires Oracle NoSQL Database to examine all the data in the store.

For a description of using indexes with non-scalar data types, see Indexing Non-Scalar Data Types (page 70).

DROP INDEX

To delete an index definition from the store, use a `DROP INDEX` statement. Its form when deleting an index is:

```
DROP INDEX [IF EXISTS] index-name ON table-name
```

where:

- `IF EXISTS` is optional, and it causes the `DROP INDEX` statement to be ignored if an index by that name does not exist. If this phrase is not specified, and an index using the specified name does not exist, then the `DROP INDEX` statement will fail with an error.
- *index-name* is the name of the index you want to drop.
- *table-name* is the name of the table containing the index you want to delete.

For example, if table `Users` has an index called `surnameIndex`, then you can delete it using the following statement:

```
DROP INDEX IF EXISTS surnameIndex ON Users
```

DESCRIBE AS JSON TABLE

You can retrieve a JSON representation of a table by using the `DESCRIBE AS JSON TABLE` statement:

```
DESCRIBE AS JSON TABLE table_name [(field-name, field-name2, ...)]
```

or

```
DESC AS JSON TABLE table_name [(field-name, field-name2, ...)]
```

where:

- *table_name* is the name of the table you want to describe.
- *field-name* is 0 or more fields defined for the table that you want described. If specified, the output is limited to just the fields listed here.

  Map and Array fields support the use of ELEMENTOF() to restrict the JSON representation to just the map or array element.

DESCRIBE AS JSON INDEX

You can retrieve a JSON representation of an index by using the `DESCRIBE AS JSON INDEX` statement:

```
DESCRIBE AS JSON INDEX index_name ON table_name
```

where:

- *index_name* is the name of the index you want to describe.
- *table_name* is the name of the table to which the index is applied.

SHOW TABLES

You can retrieve a list of all tables currently defined in the store using the `SHOW TABLES` statement:

```
SHOW [AS JSON] TABLES
```

where *AS JSON* is optional and causes the resulting output to be JSON-formatted.

SHOW INDEXES

You can retrieve a list of all indexes currently defined for a table using the `SHOW INDEXES` statement:

```
SHOW [AS JSON] INDEXES ON table_name
```

where:

- *AS JSON* is optional and causes the resulting output to be JSON-formatted.
- *table_name* is the name of the table for which you want to list all the indexes.

Appendix B. Third Party Licenses

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