# *Oracle*

# *Getting Started with NoSQL Database Python Driver*

*12c Release 1*

**Library Version 12.1.3.3**

ORACLE

NOSQL DATABASE

## Legal Notice

*Published 9/30/2015*

# Table of Contents

Preface ........ vi
- Conventions Used in This Book ........ vi

1. Developing for Oracle NoSQL Database ........ 1
   - Installing the Driver ........ 2
   - Using the Proxy Server ........ 2
   - The nosqldb Python Module ........ 3
   - Connecting to the Store ........ 3
     - Connecting to a Secure Store ........ 4
     - Automatically Starting the Proxy Server ........ 5
     - The StoreConfig Class ........ 6
     - The ProxyConfig Class ........ 7
   - Working with a Secured Store ........ 7
     - Configuring SSL ........ 8
       - Identifying the Trust Store ........ 8
       - Setting the SSL Transport Property ........ 8
     - Setting the Security Properties for a Proxy Server ........ 8
2. Introduction to Oracle KVLite ........ 10
   - Starting KVLite ........ 10
   - Stopping and Restarting KVLite ........ 11
   - Verifying the Installation ........ 11
   - kvlite Utility Command Line Parameter Options ........ 12
3. Introducing Oracle NoSQL Database Tables and Indexes ........ 14
   - Defining Tables ........ 14
     - Executing DDL Statements using the CLI ........ 15
     - Supported Table Data Types ........ 15
     - Record Fields ........ 16
     - Defining Child Tables ........ 17
     - Table Evolution ........ 18
   - Creating Indexes ........ 18
     - Indexable Field Types ........ 19
4. Primary and Shard Key Design ........ 21
   - Primary Keys ........ 21
     - Data Type Limitations ........ 22
     - Partial Primary Keys ........ 22
     - Shard Keys ........ 22
   - Row Data ........ 23
5. Writing and Deleting Table Rows ........ 26
   - Write Exceptions ........ 26
   - Writing Rows to a Table in the Store ........ 26
     - Writing Rows to a Child Table ........ 27
     - Other put Operations ........ 28
   - Deleting Rows from the Store ........ 29
     - Using multi_delete() ........ 29
6. Reading Table Rows ........ 31
   - Read Exceptions ........ 31
   - Retrieving a Single Row ........ 32

Retrieve a Child Table ..... 33
Using multi_get() ..... 34
Iterating over Table Rows ..... 36
Specifying Field Ranges ..... 38
Iterating with Nested Tables ..... 40
Reading Indexes ..... 43
7. Using Data Types ..... 46
Using Arrays ..... 46
Using Binary ..... 47
Using Enums ..... 48
Using Fixed Binary ..... 49
Using Maps ..... 50
Using Embedded Records ..... 51
8. Indexing Non-Scalar Data Types ..... 53
Indexing Arrays ..... 53
Indexing Maps ..... 54
Indexing by Map Keys ..... 54
Indexing by Map Values ..... 56
Indexing by a Specific Map Key Name ..... 57
Indexing by Map Key and Value ..... 59
Indexing Embedded Records ..... 61
9. Using Row Versions ..... 63
10. Consistency Guarantees ..... 65
Specifying Consistency Policies ..... 65
Using Simple Consistency ..... 66
Using Time-Based Consistency ..... 67
Using Version-Based Consistency ..... 68
11. Durability Guarantees ..... 72
Setting Acknowledgment-Based Durability Policies ..... 72
Setting Synchronization-Based Durability Policies ..... 73
Setting Durability Guarantees ..... 74
12. Executing a Sequence of Operations ..... 77
Sequence Errors ..... 77
Creating a Sequence ..... 77
Executing a Sequence ..... 80
A. Table Data Definition Language Overview ..... 82
Name Constraints ..... 82
DDL Comments ..... 83
CREATE TABLE ..... 83
Field Definitions ..... 84
Supported Data Types ..... 84
Field Constraints ..... 86
CHECK ..... 86
COMMENT ..... 86
DEFAULT ..... 87
NOT NULL ..... 87
Table Creation Examples ..... 87
Modify Table Definitions ..... 88
ALTER TABLE ADD field ..... 88

ALTER TABLE DROP field ........ 88
DROP TABLE ........ 89
CREATE INDEX ........ 89
DROP INDEX ........ 90
DESCRIBE AS JSON TABLE ........ 90
DESCRIBE AS JSON INDEX ........ 91
SHOW TABLES ........ 91
SHOW INDEXES ........ 91
B. Proxy Server Reference ........ 92
Securing Oracle NoSQL Database Proxy Server ........ 93
Trouble Shooting the Proxy Server ........ 96
C. Third Party Licenses ........ 99

# 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 Python.

### 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`.

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:

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:

### 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 Python drivers that are provided for the product. In addition to the Python drivers, several other drivers are also available. They are:

1. Java Table Driver
2. Java Key/Value Driver
3. C Table Driver
4. C Key/Value 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 Python 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 nosqldb driver supports Python 2.6 and 2.7.

## Installing the Driver

To install the nosqldb driver, as well as the required Java proxy server, use pip:

```
pip install nosqldb
```

The full nosqldb package source with examples and tests can be found at https://pypi.python.org/pypi/nosqldb.

The nosqldb driver depends on the Python Thrift package. Installation of the nosqldb driver using pip should resolve that dependency, but if necessary you can install Python Thrift yourself using:

```
pip install thrift
```

## Using the Proxy Server

The proxy server is a Java application that accepts network traffic from the Python Table driver, translates it into requests that the Oracle NoSQL Database store can understand, and then forwards the translated request to the store. The proxy also provides the reverse translation service by interpreting store responses and forwarding them to the client.

The proxy server can run on any network-accessible machine. It has minimal resource requirements and, in many cases, can run on the same machine as the client code is running.

Before your Python client can access the store, the proxy server must be running. It requires the following jar files to be in its class path, either by using the `java -cp` command line option, or by using the `CLASSPATH` environment variable:

- kvclient.jar
- kvproxy.jar

### Note

The proxy server, kvclient and their dependencies reside in the `<python-site-packages-directory>/nosqldb/kvproxy/lib` directory.

The proxy server itself is started using the `oracle.kv.proxy.KVProxy` command. At a minimum, the following information is required when you start the proxy server:

- `-helper-hosts`

  A list of one or more host:port pairs representing Oracle NoSQL Database storage nodes that the proxy server can use to connect to the store.

- `-port`

  The port where your client code can connect to this instance of the proxy server.

- `-store`

  The name of the store to which the proxy server is connecting.

A range of other command line options are available. In particular, if you are using the proxy server with a secure store, you must provide authentication information to the proxy server. (Note that the proxy server itself connects to a single store using a single user credential. If multiple stores or users are required, then multiple proxy servers must be used.) In addition, you will probably have to identify a store name to the proxy server. For a complete description of the proxy server and its command line options, see Proxy Server Reference (page 92).

The examples provided in this guide were written to work with a proxy server that is connected to a `kvlite` instance which was started with default values. The command line call used to start the proxy server was:

```
nohup java oracle.kv.proxy.KVProxy -port 7010 \
-helper-hosts localhost:5000 -store kvstore
```

Usage of kvlite is described in Introduction to Oracle KVLite (page 10).

## The nosqldb Python Module

All of the classes and methods that you use to perform Oracle NoSQL Database store access are contained in the `nosqldb` Python module.

The nosqldb module makes use of the standard Python logging facility. It uses the "nosqldb" logger, not the root logger. The examples in this document take advantage of the logging facility by issuing DEBUG and ERROR messages through it. Logging is also sent to stdout using the following setup function:

```
import logging

...

# set logging level to debug and log to stdout
def setup_logging():
    logger = logging.getLogger("nosqldb")
    logger.setLevel(logging.DEBUG)

    logger = logging.StreamHandler(sys.stdout)
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('\t%(levelname)s - %(message)s')
    logger.setFormatter(formatter)
    rootLogger.addHandler(logger)
```

You can also set logging levels using the `StoreConfig.change_log()` method. You can turn off logging completely using `StoreConfig.turn_off_log()`.

## Connecting to the Store

To perform store operations, you must establish a network connection between your client code and the store. There are three pieces of information that you must provide:

- 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 running in the store. Multiple nodes can be identified. 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.
- Identify the host and port where the proxy is running. You also do this using the configuration object.

If you are connecting to a secured store, you must also provide some authentication information. This is described in Setting the Security Properties for a Proxy Server (page 8).

For example, suppose you have an Oracle NoSQL Database store named "kvstore" and it has a node running on n1.example.org at port 5000. Further, suppose you are running your proxy on the localhost using port 7010. Then you would open and close a connection to the store in the following way:

```
from nosqldb import ConnectionException
from nosqldb import Factory
from nosqldb import StoreConfig

import logging
import sys

storehost = "n1.example.org:5000"
proxy = "localhost:7010"

# configure and open the store
def open_store():
    try:
        kvstoreconfig = StoreConfig('kvstore', [storehost])
        return Factory.open(proxy, kvstoreconfig)
    except ConnectionException, ce:
        logging.error("Store connection failed.")
        logging.error(ce.message)
        sys.exit(-1)
```

`Factory.open()` returns a `Store` class object, which you use to perform most operations against your store. When you are done with this handle, close it using the `close()` method:

```
store = open_store()

...
# Do store operations here
...

store.close()
```

## Connecting to a Secure Store

If you are using a secure store, then your proxy server must first be configured to authenticate to the store. See Securing Oracle NoSQL Database Proxy Server (page 93) for details.

Once your proxy server is capable of accessing the secure store, you must at a minimum indicate which user your driver wants to authenticate as when it performs store access. To do this, use the `StoreConfig.set_user()` method.

For more information on using secure stores, see Working with a Secured Store (page 7).

```
# configure and open the store
def open_store():
    try:
        kvstoreconfig = StoreConfig('kvstore', [storehost])
        kvstoreconfig.set_user("pythonapp-user")
        return Factory.open(proxy, kvstoreconfig)
    except ConnectionException, ce:
        logging.error("Store connection failed.")
        logging.error(ce.message)
        sys.exit(-1)
```

## Automatically Starting the Proxy Server

If it is not already running, your client code will automatically start the proxy server on the local host when it opens the store so long as it can locate the `kvclient.jar` and `kvproxy.jar` files. These are automatically installed when you install the nosqldb driver, so you should not normally need to do anything extra in order to have the driver automatically start the proxy server.

However, if you installed the nosqldb driver in a non-standard location, or if you want to override the default jar files installed on your system, then you can explicitly tell the driver where these jar files are located:

1. If they are specified as parameters to the `ProxyConfig` constructor, then that location is used.

2. If that information is not specified to the constructor, then it is taken from the `KVSTORE_JAR` and `KVPROXY_JAR` environment variables.

3. If neither of the above methods are used, then the driver uses the default jar files, which are installed in `<python-site-packages-dir>/nosqldb/kvproxy/lib`

In the following example, two environment variables are defined like this:

```
export KVSTORE_JAR="/d1/nosqldb-x.y.z/kvproxy/lib/kvclient.jar"
export KVPROXY_JAR="/d1/nosqldb-x.y.z/kvproxy/lib/kvproxy.jar"
```

Because these environment variables are set, the `ProxyConfig` constructor will automatically use them as the location for the jar files.

```
# configure and open the store
def open_store():

    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvproxyconfig = ProxyConfig()
    return Factory.open(proxy, kvstoreconfig, kvproxyconfig)
```

Be aware that if your proxy is connecting to a secure store, you also must indicate which user to authenticate as, and you must indicate where the security properties file is located on the host where the proxy server is running.

```
# configure and open the store
def open_store():

    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvstoreconfig.set_user("pythonapp-user")
    kvproxyconfig = ProxyConfig()
    kvproxyconfig.set_security_properties_file("/etc/proxy/sec.props")

    return Factory.open(proxy, kvstoreconfig, kvproxyconfig)
```

For information on configuring your proxy server to connect to a secure store, see Securing Oracle NoSQL Database Proxy Server (page 93).

## The StoreConfig Class

The `StoreConfig` class is used to describe properties about a `Store` handle. Most of the properties are optional; however, you must identify the store name and helper hosts.

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

- set_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 65).

- set_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 72).

- set_max_results()

  The number of rows buffered by iterators.

- set_read_zones()

  An array of zone names to be used as read zones. For more information on read zones, see the *Oracle NoSQL Database Administrator's Guide*.

- set_request_timeout()

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

- set_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:

  ```
  conf.set_helper_hosts(['n1.example.org:3333','n2.example.org:3333'])
  ```

- set_store_name()

  Identifies the name of the store.

- set_user()

  The name of the user you want to authenticate to the store as. This property should only be used when your proxy server is configured to connect to a secure store.

## The ProxyConfig Class

The `ProxyConfig` class is used to describe properties about the proxy server you are using to connect to the store.

The number of properties you can specify using this class are too numerous to describe here (see the *Oracle NoSQL Python Driver for Tables API Reference* for a complete listing), but the most common properties you will set are:

- set_security_props_file()

  The properties file containing the security information required to connect to a secure store. For information on secure stores and security properties, see Setting the Security Properties for a Proxy Server (page 8).

- set_kv_store_path_to_jar()

  The path where the kvstore.jar file is located. This information is only required if you are attempting to automatically start the proxy server.

- set_kv_proxy_path_to_jar()

  The path where the kvproxy.jar file is located. This information is only required if you are attempting to automatically start the proxy server.

## Working with a Secured Store

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 the proxy server connects to the store. The proxy server gets the username that it uses from the client code connecting to it. The rest of the credential information is contained in a security properties file that must be installed on the same local host as the proxy server. Be aware that the connection from your driver code to the proxy server is not secure. It is therefore recommended that you run your proxy server and your driver code on the same physical host.

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 proxy server runs in order to validate that the store that is being accessed is trustworthy.

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

## Configuring SSL

If you are using a secure store, then all communications between your proxy server 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 a proxy server.

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

Your proxy server 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.

### Setting the SSL Transport Property

In addition to identifying the location of the `client.trust` file, you must also tell your proxy server to use the SSL transport. You do this by setting the `oracle.kv.transport` property.

## Setting the Security Properties for a Proxy Server

When an Oracle NoSQL Database secure store is installed, a file is generated called `client.security`. This file contains security properties that are needed by your proxy server. Copy this file to the machine that will run your proxy server, along with your `client.trust` file and the `login.passwd` file. All of these files were created during the installation process.

You may need to edit your `client.security` file to adjust the location of files for the local host. When you get done, `client.security` should look something like this:

```
oracle.kv.auth.username=root
oracle.kv.auth.pwdfile.file=/security/login.passwd
oracle.kv.transport=ssl
oracle.kv.ssl.trustStore=/security/client.trust
oracle.kv.ssl.protocols=TLSv1.2,TLSv1.1,TLSv1
oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
```

Once these files are in place and are edited correctly, when your client code creates a store connection, it must identify both the location of the `client.security` file *on the disk where the proxy server is running*, as well as the username the client wants to authenticate as. See Connecting to the Store (page 3) for an example of how this is done.

# 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 Working with a Secured Store (page 7) for information on configuring your proxy server 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 82) 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 21) 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 `Store.execute()` or `Store.execute_sync()` 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 Connecting to the Store (page 3).

For example:

```
...
```

```
    ### Store handle configuration and open skipped for brevity
    ...

    try:
        ddl = """CREATE TABLE myTable (
            id STRING,
            description STRING,
            count INTEGER,
            percentage FLOAT,
            PRIMARY KEY (item)
        )"""
        store.execute_sync(ddl)
        logging.debug("Table creation succeeded")
    except IllegalArgumentException, iae:
        logging.error("DDL failed.")
        logging.error(iae.message)
```

## 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

For the Python driver, these datatypes are handled in the following way:

| Oracle NoSQL Database Datatype | Python Datatype |
|---|---|
| Array | Python array. See Using Arrays (page 46). |
| Binary | Base64 encoded buffer. See Using Binary (page 47). |
| Boolean | Python boolean |
| Double | Python long |
| Enum | Python string. See Using Enums (page 48). |
| Fixed Binary | Base64 encoded buffer. See Using Fixed Binary (page 49). |
| Float | Python float |
| Integer | Python int |
| Long | Python long |
| Map | Python dictionary. See Using Maps (page 50). |
| Records | Python dictionary. See Using Embedded Records (page 51). |
| String | Python string |

## Record Fields

As described in Defining Child Tables (page 17), 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 33) 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 88) 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 19) 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 89) 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 90) 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 `Store.execute()` method.

```
...
### Store handle configuration and open skipped for brevity
...

    try:
        ddl = """CREATE INDEX inventoryIdx on
                 myInventory.itemDetails(inventoryCount)"""
        store.execute_sync(ddl)
        logging.debug("Index creation succeeded")
    except IllegalArgumentException, iae:
        logging.error("DDL failed.")
        logging.error(iae.message)
```

## 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 53) 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 31). We show how to delete multiple rows at a time in Using multi_delete() (page 29).

## 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 77) 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.

# 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 errors that you should handle whenever you perform a write operation to the store. Some of the more common errors 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 errors.

The first of these is `DurabilityException`. This error indicates that the operation cannot be completed because the durability policy cannot be met. For more information, see Durability Guarantees (page 72).

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.

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 error occurred which is neither a problem with durability nor a problem with the request timeout. Your only recourse here is to either log the error and move along, or retry the operation.

## Writing Rows to a Table in the Store

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

Remember that you can only write data to a table after it has been added to the store. See Introducing Oracle NoSQL Database Tables and Indexes (page 14) for details.

To write a row to a table in the store:

1. Create a store handle and open it.
2. Use a dictionary to describe the row. Each key in the dictionary must correspond to a field name that has been declared for the table to which you will write the row.

3. Use the `Store.put()` method to write the row to the store. This method has two required arguments. The first identifies the table to which you want to write the row. The second accepts the dictionary you constructed in the previous step.

For example:

```
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import StoreConfig

import logging
import os

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

kvstoreconfig = StoreConfig('kvstore', [kvlite])
store = Factory.open(proxy, kvstoreconfig)

row_d = { 'item' : 'bolts',
           'description' : "Hex head, stainless",
           'count' : 5,
           'percentage' : 0.2173913}
try:
    store.put("myTable", row_d)
    logging.debug("Store write succeeded.")
except IllegalArgumentException, iae:
    logging.error("Could not write table.")
    logging.error(iae.message)

store.close()
```

## 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 17) we showed how to create a child table. To write data to that table, do this:

```
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import StoreConfig

import logging
import os
```

```
# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

kvstoreconfig = StoreConfig('kvstore', [kvlite])
store = Factory.open(proxy, kvstoreconfig)

parent_d = {'itemCategory' : 'Bolts',
            'description'  : 'Metric & US sizes'}

try:
    store.put("myInventory", parent_d)
    logging.debug("Store write succeeded.")
except IllegalArgumentException, iae:
    logging.error("Could not write table.")
    logging.error(iae.message)

child_d = {'itemCategory'    : 'Bolts',
           'itemSKU'         : '1392610',
           'itemDescription' : "1/4-20 x 1/2 Grade 8 Hex",
           'price'           : 11.99,
           'inventoryCount'  : 1457

child_row= Row(child_d)
try:
    store.put("myInventory.itemDetails", child_row)
    logging.debug("Store write succeeded.")
except IllegalArgumentException, iae:
    logging.error("Could not write table.")
    logging.error(iae.message)


store.close()
```

## Other put Operations

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

- `Store.put_if_absent()`

  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.

- `Store.put_if_present()`

  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.

- `Store.put_if_version()`

This method will put the row only if the value matches the supplied version information. For more information, see Using Row Versions (page 63).

## Deleting Rows from the Store

You delete a single row from the store using the `Store.delete()` method. Rows are deleted based on a dictionary that defines the full primary key for the row that you want to delete. You can also require a row to match a specified version before it will be deleted. To do this, use the `Store.delete_if_version()` method. Versions are described in Using Row Versions (page 63).

When you delete a row, you must handle the same errors as occur when you perform any write operation on the store. See Write Exceptions (page 26) for a high-level description of these errors.

```
try:
    # To delete a table row, just include a dictionary
    # that contains all the fields needed to create
    # the primary key.
    primary_key_d = {"item" : "bolts"}
    ret = store.delete("myTable", primary_key_d)
    if ret[0]:
        logging.debug("Row deletion succeeded")
    else:
        logging.debug("Row deletion failed.")
except IllegalArgumentException, iae:
    logging.error("Row deletion failed.")
    logging.error(iae.message)
```

### Using multi_delete()

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 `Store.multi_delete()` 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:

```
try:
    primary_key_d = {'itemType'     : 'Hats',
                     'itemCategory' : 'baseball',
                     'itemClass'    : 'longbill'}

    ret = store.multi_delete("myTable", primary_key_d)
    if ret > 0:
        logging.debug("%s rows deleted" % ret)
    else:
        logging.debug("No rows deleted.")
except IllegalArgumentException, iae:
    logging.error("Row deletion failed.")
    logging.error(iae.message)
```

# 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 `Store.get()` method.
2. Retrieve rows associated with a shard key (which is based on at least part of your primary keys) using the `Store.multi_get()` method.
3. Retrieve table rows that share a shard key, or an index key, using the `Store.table_iterator()` 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 errors can occur when you attempt a read operation in the store. The first of these is `ConsistencyException`. This error indicates that the operation cannot be completed because the consistency policy cannot be met. For more information, see Consistency Guarantees (page 65).

The second error 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 error 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().

## Retrieving a Single Row

To retrieve a single row from the store:

1. Create a store handle and open it.
2. Construct a Python dictionary. primary key. Each name/value pair in the dictionary must correspond to the primary key and value for the row that you want to retrieve. In this case, the full primary key must be present in the dictionary.
3. Retrieve the row using `Store.get()`. This performs the store read operation.
4. The retrieved row is a Python dictionary. Individual items in the dictionary can be retrieved as you would for any Python dictionary.

For example, in Writing Rows to a Table in the Store (page 26) we showed a trivial example of storing a table row to the store. The following trivial example shows how to retrieve that row.

```
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import StoreConfig

import logging
import os
import sys

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

# set logging level to debug and log to stdout
def setup_logging():
    rootLogger = logging.getLogger()
    rootLogger.setLevel(logging.DEBUG)

    logger = logging.StreamHandler(sys.stdout)
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('\t%(levelname)s - %(message)s')
    logger.setFormatter(formatter)
    rootLogger.addHandler(logger)

# configure and open the store
def open_store():
    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    return Factory.open(proxy, kvstoreconfig)
```

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tItem: %s" % row['item']
            print "\tDescription: %s" % row['description']
            print "\tCount: %s" % row['count']
            print "\tPercentage: %s" % row['percentage']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

def do_store_ops(store):
    try:
        primary_key_d = {"item" : "bolts"}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            display_row(row)
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

if __name__ == '__main__':

    setup_logging()
    store = open_store()
    do_store_ops(store)
    store.close()
```

## Retrieve a Child Table

In Writing Rows to a Child Table (page 27) 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:

```
...

def do_store_ops(store):
    try:
        primary_key_c = {"item" : "bolts",
                         "itemSKU" : "1392610"}

        row = store.get("myInventory.itemDetails", primary_key_c)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            print row
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

For information on how to iterate over nested tables, see Iterating with Nested Tables (page 40).

## Using multi_get()

`Store.multi_get()` 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 `Store.multi_get()` 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.

```
...

def display_row(row):
    try:
            print "Retrieved row:"
            print "\tItem Type: %s" % row['itemType']
            print "\tCategory: %s" % row['itemCategory']
            print "\tClass: %s" % row['itemClass']
            print "\tSize: %s" % row['itemSize']
            print "\tColor: %s" % row['itemColor']
            print "\tPrice: %s" % row['price']
            print "\tInventory Count: %s" % row['inventoryCount']
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

def do_store_ops(store):
    try:
        shard_key_d = {"itemType" : "Hats",
                       "itemCategory" : "baseball",
                       "itemClass" : "longbill"}

        row_list =
            store.multi_get("myTable",   # table name
                            False,       # Retrieve only keys?
                            shard_key_d) # partial primary key
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

Notice in the previous example that `Store.multi_get()` returns the table rows in a simple Python list. To display the rows, you simply iterate over the list in the same way you would any Python list.

## Iterating over Table Rows

`Store.table_iterator()` provides non-atomic table iteration.

`Store.table_iterator()` 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` object that lets you specify:
  - A `FieldRange` object, which defines a range of values to be retrieved for the specified key.
  - A list of parent and ancestor tables to include in the iteration.
- A `TableIteratorOptions` object, which allows you to specify an iteration direction, the maximum number of results to return for each retrieval batch, and a `ReadOptions` class. This class allows you specify a consistency policy for the operation, as well as an upper bound on the amount of time that the operation is allowed to take. Consistency policies are described in Consistency Guarantees (page 65).

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 `Store.table_iterator()` as follows. Note that this simple example can also be accomplished using the `Store.multi_get()` method. If you have a complete shard key, and if the entire results set will fit in memory, then `multi_get()` will perform much better than `table_iterator()`. However, if the results set cannot fit entirely in memory, or if you do not have a complete shard key, then `table_iterator()` is the better choice. Note that reads performed using `table_iterator()` are non-atomic, which may have ramifications if you are performing a long-running iteration over records that are being updated.

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tType: %s" % row['itemType']
            print "\tCategory: %s" % row['itemCategory']
            print "\tClass: %s" % row['itemClass']
            print "\tColor: %s" % row['itemColor']
            print "\tSize: %s" % row['itemSize']
            print "\tPrice: %s" % row['price']
            print "\tInventory Count: %s" % row['inventoryCount']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)


def do_store_ops(store):

    key_d = {'itemType' : 'Hats'}

    try:
        row_list = store.table_iterator("myTable", key_d, False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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 the `FieldRange` class, which you provide to the method you are using to perform the multi-read operation using the `MultiRowOptions` class. This class requires the name of the primary key field for which you want to set the range, as well the range values, including whether they are inclusive.

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

In this example, we use `TableIterator`, but we could just as easily use this range on any multi-row read operation, such as the `Store.multi_get()` method.

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tSurname: %s" % row['surname']
            print "\tFamiliar Name: %s" % row['familiarName']
            print "\tUser ID: %s" % row['userID']
            print "\tPhone: %s" % row['phonenumber']
            print "\tAddress: %s" % row['address']
            print "\tEmail: %s" % row['email']
            print "\tDate of Birth: %s" % row['dateOfBirth']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)


def do_store_ops(store):

    key_d = {'surname' : 'Smith'}

    field_range = FieldRange({
                         ONDB_FIELD : "familiarName",
                         ONDB_START_VALUE : "Bob",
                         ONDB_END_VALUE : "Patricia",
                         # These next two are the default values,
                         # so are not really needed.
                         ONDB_START_INCLUSIVE : True,
                         ONDB_END_INCLUSIVE : True
                         })

    mro = MultiRowOptions({ONDB_FIELD_RANGE : field_range})
    try:
        row_list = store.table_iterator("myTable", key_d, False, mro)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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 using a list of table names, which is then provided to the `MultiRowOpetions` object's `ONDB_INCLUDED_TABLES` property.

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

```
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import StoreConfig
from nosqldb import ONDB_INCLUDED_TABLES

import logging
import os
import sys

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

# set logging level to debug and log to stdout
```

```
def setup_logging():
    rootLogger = logging.getLogger()
    rootLogger.setLevel(logging.DEBUG)

    logger = logging.StreamHandler(sys.stdout)
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('\t%(levelname)s - %(message)s')
    logger.setFormatter(formatter)
    rootLogger.addHandler(logger)

# configure and open the store
def open_store():
    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    return Factory.open(proxy, kvstoreconfig)

def display_row(row):
    try:
        ## Our code must track which table we are displaying.
        ## Use get_table_name() for this purpose.

        if row.get_table_name() == 'prodTable':
            print "\nType: %s" % row['prodType']
            print "Description: %s" % row['typeDescription']
        elif row.get_table_name() == 'prodTable.prodCategory':
            print "\tCategory: %s" % row['categoryName']
            print "\tDescription: %s" % row['categoryDescription']
        else:
            print "\t\tSKU: %s" % row['itemSKU']
            print "\t\tDescription: %s" % row['itemDescription']
            print "\t\tPrice: %s" % row['itemPrice']
            print "\t\tVendor UID: %s" % row['vendorUID']
            print "\t\tInventory Count: %s" % row['inventoryCount']
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

def do_store_ops(store):
    try:
        key_d = {}

        ## Identify the child tables to include in the retrieval.
        incTables = ["prodTable.prodCategory",
                     "prodTable.prodCategory.item"]

        mro = {ONDB_INCLUDED_TABLES : incTables}

        row_list = store.table_iterator("prodTable", key_d, False,
                mro)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

if __name__ == '__main__':

    setup_logging()
    store = open_store()
    do_store_ops(store)
    store.close()
```

## Reading Indexes

You use `Store.index_iterator()` to retrieve table rows using a table's indexes. Just as when you use `table_terator` 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.

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 scan through that index, do the following:

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tSurname: %s" % row['surname']
            print "\tFamiliar Name: %s" % row['familiarName']
            print "\tUser ID: %s" % row['userID']
            print "\tPhone: %s" % row['phonenumber']
            print "\tAddress: %s" % row['address']
            print "\tEmail: %s" % row['email']
            print "\tDate of Birth: %s" % row['dateOfBirth']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)


def do_store_ops(store):

    key_d = {}

    try:
        row_list = store.index_iterator("myTable", "DoB",
                                        key_d, False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

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 constructing 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 object 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 the `FieldRange` object:

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tSurname: %s" % row['surname']
            print "\tFamiliar Name: %s" % row['familiarName']
            print "\tUser ID: %s" % row['userID']
            print "\tPhone: %s" % row['phonenumber']
            print "\tAddress: %s" % row['address']
            print "\tEmail: %s" % row['email']
            print "\tDate of Birth: %s" % row['dateOfBirth']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)


def do_store_ops(store):
```

```
key_d = {}

field_range = FieldRange({
                         ONDB_FIELD : "dateOfBirth",
                         ONDB_START_VALUE : "1994-05-01",
                         ONDB_END_VALUE : "1994-05-30",
                         # These next two are the default values,
                         # so are not really needed.
                         ONDB_START_INCLUSIVE : True,
                         ONDB_END_INCLUSIVE : True
                         })

mro = MultiRowOptions({ONDB_FIELD_RANGE : field_range})

try:
    row_list = store.index_iterator("myTable", "DoB",
                                     key_d, False, mro)
    if not row_list:
        logging.debug("Table retrieval failed")
    else:
        logging.debug("Table retrieval succeeded.")
        for r in row_list:
            display_row(r)
except IllegalArgumentException, iae:
    logging.error("Table retrieval failed.")
    logging.error(iae.message)
```

# Chapter 7. Using Data Types

Many of the types that Oracle NoSQL Database offers are easy to use (such as integers and strings). Examples of their usage has been scattered throughout this manual. However, some types are a little more complicated, and so their usage may not be obvious. This chapter briefly shows how to use Arrays, Maps, Records, Enums and Binary data 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 86) for more details.

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

To write the array:

```
    row_d = {'uid' : 0,
             'myArray' : ["One", "Two", "Three"]
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

To retrieve and use the array:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            myArray = row['myArray']
            for m in myArray:
                print m
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

## Using Binary

You can declare a field as binary using the `BINARY` statement. You then read and write the field value as a base64 encoded buffer.

If you want to store a large binary object, then you should use the LOB APIs rather than a binary field, which are only available using the Java Key/Value API. 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 49) 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 binary field, use the `Store.encode_base_64()` method to encode the data before writing it to the store.

```
    iFile = open("image.jpg")
    image = store.encode_base_64(iFile.read())
    iFile.close()

    row_d = {'uid' : 0,
             'myByteArray' : image
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

To read the binary field, retrieve it as you would any data field. Use `Store.decode_base_64()` to decode the data before writing it to disk.

For example:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            image = store.decode_base_64(row['myByteArray'])
            iFile = open("out.jpg", 'w')
            iFile.write(image)
            iFile.close()
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

## 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 87) for more information.

Enum values are handled as strings.

To write the enum:

```
    row_d = {'uid' : 0,
             'myEnum' : 'Pears'
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

To read the enum:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            myEnum = row['myEnum']
            print "myEnum: %s" % myEnum
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

## 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 a base64 encoded buffer. However, if the buffer does not equal the specified size, then `IllegalArgumentException` is thrown when you attempt to write the field.

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 Oracle NoSQL API Large Object API.

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 47) 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,
    myFixedByteArray BINARY(10),
    PRIMARY KEY (uid)
)
```

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

To write the byte array:

```
    b64buffer = store.encode_base_64('1234567890')
    row_d = {'uid' : 0,
             'myFixedByteArray' : b64buffer
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

To read the fixed binary field, use `Store.decode_base_64()`:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            b64buffer = row['myFixedByteArray']
            print store.decode_base_64(b64buffer)
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

## 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 86) for more information.

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

To write the map:

```
    mmap = {"field1" : 1,
            "field2" : 2,
            "field3" : 3}

    row_d = {'uid' : 0,
             'myMap' : mmap
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

To read map field2:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            ## prints '2'
            print row['myMap']['field2']
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

## 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 86) for more information.

To write the embedded record, define it as a Python map:

```
    mrec = {"firstField" : "An embedded record",
            "secondField" : 3388}

    row_d = {'uid' : 0,
             'myRecord' : mrec
            }
    try:
        store.put("myTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)
```

Then, you can read the field in the usual way:

```
    try:
        primary_key_d = {"uid" : 0}
        row = store.get("myTable", primary_key_d)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            print "firstField: %s" % row['myRecord']['firstField']
            print "secondField: %s" % row['myRecord']['secondField']
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)
```

# Chapter 8. Indexing Non-Scalar Data Types

We describe how to index scalar data types in Creating Indexes (page 18), and we show how to read using indexes in Reading Indexes (page 43). 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 89) 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, create a key that identifies the array field and value that you want to retrieve.

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 key object. 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.

For example:

```
    try:
        key_d = {"testArray" : ["One"]}
        row_list = store.index_iterator("myArrayTable",
                                        "arrayFieldIndex",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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:

```
...

def writeStore(store, row_d):

    try:
        store.put("myMapTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

...

def populateTable(store):

   row_d = {'uid' : 12345,
             'testMap' : {'field1' : 1, 'field2' : 2, 'field3' : 3}
            }
    writeStore(store, row_d)


    row_d = {'uid' : 12,
             'testMap' : {'field1' : 1, 'field2' : 2}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 666,
             'testMap' : {'field1' : 1, 'field3' : 4}
            }
    writeStore(store, row_d)
```

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

Note that we use a simple Python dictionary to represent the map. Because the index we are using is based on the map's key, we can just use `None` for the map's value.

```
def readStore(store):
    try:
        key_d = {"testMap" : {'field3' : None}}
        row_list = store.index_iterator("myMapTable",
                                        "mapKeyIndex",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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:

```
...

def writeStore(store, row_d):

    try:
        store.put("myMapTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

...

def populateTable(store):

   row_d = {'uid' : 12345,
             'testMap' : {'field1' : 1, 'field2' : 2, 'field3' : 3}
            }
    writeStore(store, row_d)


    row_d = {'uid' : 12,
             'testMap' : {'field1' : 1, 'field2' : 2}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 666,
             'testMap' : {'field1' : 1, 'field3' : 4}
            }
    writeStore(store, row_d)
```

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".

Notice in the following example that we use the special string "[]" for the index key's field value. The field name must be that string or we will not access the proper index.

```
def readStore(store):
    try:
        key_d = {"testMap" : {"[]" : 2}}
        row_list = store.index_iterator("myMapTable",
                                        "mapElementIndex",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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:

```
...

def writeStore(store, row_d):

    try:
        store.put("myMapTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

...

def populateTable(store):

   row_d = {'uid' : 12345,
             'testMap' : {'field1' : 1, 'field2' : 2, 'field3' : 3}
            }
    writeStore(store, row_d)


    row_d = {'uid' : 12,
             'testMap' : {'field1' : 1, 'field2' : 2}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 666,
             'testMap' : {'field1' : 1, 'field3' : 4}
            }
    writeStore(store, row_d)
```

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".

```
def readStore(store):
    try:
        key_d = {"testMap" : {"field3" : 3}}
        row_list = store.index_iterator("myMapTable",
                                        "mapField3Index",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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:

```
...

def writeStore(store, row_d):
    try:
        store.put("myMapTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

...

def populateTable(store):

   row_d = {'uid' : 12345,
             'testMap' : {'field1' : 1, 'field2' : 2, 'field3' : 3}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 12,
             'testMap' : {'field1' : 1, 'field2' : 2}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 666,
             'testMap' : {'field1' : 1, 'field3' : 4}
            }
    writeStore(store, row_d)
```

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 based on this kind of an index, you must provide:

- the special string '[]' with the desired map value; and
- the field name with `None` for a map value.

You do this in Python in the following way:

```
def readStore(store):
    try:
        key_d = {"testMap" : {"[]" : 3, "field3" : None}}
        row_list = store.index_iterator("myMapTable",
                                        "mapKeyValueIndex",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

## 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:

```
def writeStore(store, row_d):

    try:
        store.put("myRecordTable", row_d)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

def populateTable(store):
    row_d = {'uid' : 12345,
             'myRecord' : {'firstField' : 'String field for 12345',
                           'secondField' : 3388}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 345,
             'myRecord' : {'firstField' : 'String field for 345',
                           'secondField' : 3388}
            }
    writeStore(store, row_d)

    row_d = {'uid' : 111,
             'myRecord' : {'firstField' : 'String field for 111',
                           'secondField' : 12}
            }
    writeStore(store, row_d)
```

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.)

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

```
def readStore(store):
    try:
        key_d = {"myRecord" : {'secondField' : 3388}}
        row_list = store.index_iterator("myRecordTable",
                                        "recordFieldIndex",
                                        key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                print r
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

# 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, `Store.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 `Store.put_if_version()` or `Store.delete_if_version()` 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 65).

Versions are handled as Python byte arrays. There is no class or other data type used to manage them. In some cases they are accessed by special methods, such as `Row.get_version()`.

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

```
def do_store_ops(store):

    key_d = {}

    try:
        row_list = store.index_iterator("myTable", "DoB", key_d,
                                        False)
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                version = r.get_version()

                ###
                ### do work on the row here
                ###

                store.put_if_version("myTable", r, version)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

# 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, use one of:

- `SimpleConsistency`
- `TimeConsistency`
- `VersionConsistency`

Each of these are described in the following sections.

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 `StoreConfig.set_consistency()` 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 `ReadOptions` class instance you provide to the `Store` 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.

```
from nosqldb import Factory
from nosqldb import StoreConfig

## available consistency constants
from nosqldb import ABSOLUTE
from nosqldb import NONE_REQUIRED
from nosqldb import NONE_REQUIRED_NO_MASTER


...

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

...

# configure and open the store
def open_store():
    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvstoreconfig.set_consistency(NONE_REQUIRED)

    return Factory.open(proxy, kvstoreconfig)

...
```

## Using Simple Consistency

You can use pre-defined consistency constants to specify certain rigid consistency guarantees. There are three such instances that you can use:

1. `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. `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. `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:

```
...
### Store handle configuration and open skipped for brevity

def do_store_ops(store):
    ## Create the simple consistency guarantee to use for this
    ## store read.
    ro = ReadOptions({ONDB_CONSISTENCY : ABSOLUTE,
                      ONDB_TIMEOUT : 600})
    try:
        primary_key_d = {"item" : "bolts"}
        row = store.get("myTable", primary_key_d, ro)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            display_row(row)
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except ConsistencyException, ce:
        logging.error("Row retrieval failed due to Consistency.")
        logging.error(ce.message)
    except RequestTimeoutException, rte:
        logging.error("Row retrieval failed, exceeded timeout value.")
        logging.error(rte.message)
```

## 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 `TimeConsistency` class. This class requires the following information:

- `ONDB_PERMISSIBLE_LAG`

The number of milliseconds the replica is allowed to lag behind the master.

- ONDB_TIMEOUT

  The number of milliseconds 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.

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

```
from nosqldb import Factory
from nosqldb import StoreConfig
from nosqldb import TimeConsistency

## Required for TimeConsistency
from nosqldb import ONDB_PERMISSIBLE_LAG
from nosqldb import ONDB_TIMEOUT

...

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

...

# configure and open the store
def open_store():
    tc = TimeConsistency({ONDB_PERMISSIBLE_LAG : 2000,
                          ONDB_TIMEOUT : 4000})

    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvstoreconfig.set_consistency(tc)

    return Factory.open(proxy, kvstoreconfig)

...
```

## 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 `VersionConsistency` class. When you do this, you must provide the following information:

- `ONDB_VERSION`

  The `Version` that the read must match.

- `ONDB_TIMEOUT`

  The number of milliseconds 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.

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

```
from nosqldb import Consistency

from nosqldb import DurabilityException
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig

from nosqldb import ReadOptions
#### constant needed for ReadOptions
from nosqldb import ONDB_CONSISTENCY
from nosqldb import ONDB_VERSION_CONSISTENCY
from nosqldb import ONDB_TIMEOUT

from nosqldb import RequestTimeoutException
from nosqldb import Row
from nosqldb import StoreConfig
from nosqldb import WriteOptions
##### Constants needed for the write options
from nosqldb import ONDB_RETURN_CHOICE

from nosqldb import VersionConsistency
### Constants needed for the VersionConsistency
from nosqldb import ONDB_TIMEOUT
from nosqldb import ONDB_VERSION

import logging
import os
import sys

# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

# set logging level to debug and log to stdout
def setup_logging():
    rootLogger = logging.getLogger()
    rootLogger.setLevel(logging.DEBUG)

    logger = logging.StreamHandler(sys.stdout)
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('\t%(levelname)s - %(message)s')
    logger.setFormatter(formatter)
    rootLogger.addHandler(logger)

# configure and open the store
def open_store():
    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    return Factory.open(proxy, kvstoreconfig)

def write_row(store):
    row_d = { 'item' : 'bolts',
              'description' : "Hex head, stainless",
              'count' : 5,
              'percentage' : 0.2173913}
    row = Row(row_d)

    ## Create the write options
    wo = WriteOptions({ONDB_RETURN_CHOICE : 'VERSION'})
    try:
        matchVersion = store.put("myTable", row, wo)
        ## matchVersion is actually a tuple, the second element of
        ## which identifies the table that the row was written to.
        return matchVersion[0]
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
    except DurabilityException, de:
        logging.error("Could not write table. Durability failure.")
        logging.error(de.message)
    except RequestTimeoutException, rte:
        logging.error("Could not write table. Exceeded timeout.")
        logging.error(rte.message)
```

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.

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tItem: %s" % row['item']
            print "\tDescription: %s" % row['description']
            print "\tCount: %s" % row['count']
            print "\tPercentage: %s" % row['percentage']
            print "\n"
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

def read_row(store, matchVersion):
    vc = VersionConsistency({ONDB_VERSION : matchVersion,
                             ONDB_TIMEOUT : 1000})

    consistency = Consistency({ONDB_VERSION_CONSISTENCY: vc})


    ro = ReadOptions({ONDB_CONSISTENCY : consistency,
                      ONDB_TIMEOUT : 1000})

    try:
        primary_key_d = {"item" : "bolts"}
        row = store.get("myTable", primary_key_d, ro)
        if not row:
            logging.debug("Row retrieval failed")
        else:
            logging.debug("Row retrieval succeeded.")
            display_row(row)
    except IllegalArgumentException, iae:
        logging.error("Row retrieval failed.")
        logging.error(iae.message)
        return
    except ConsistencyException, ce:
        logging.error("Row retrieval failed due to Consistency.")
        logging.error(ce.message)
    except RequestTimeoutException, rte:
        logging.error("Row retrieval failed, exceeded timeout value.")
        logging.error(rte.message)



if __name__ == '__main__':

    setup_logging()
    store = open_store()
    matchVersion = write_row(store)
    read_row(store, matchVersion)
    store.close()
```

# 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 `Store` 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 `Store` handle:

```
from nosqldb import Durability
### Constants needed for Durability
from nosqldb import ONDB_AP_NONE
from nosqldb import ONDB_AP_SIMPLE_MAJORITY
from nosqldb import ONDB_MASTER_SYNC
from nosqldb import ONDB_REPLICA_SYNC
from nosqldb import ONDB_REPLICA_ACK
from nosqldb import ONDB_SP_SYNC
from nosqldb import ONDB_SP_NO_SYNC
##      For Durability, could use one of
##      COMMIT_SYNC, COMMIT_NO_SYNC, or
##      COMMIT_WRITE_NO_SYNC

from nosqldb import DurabilityException
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import RequestTimeoutException
from nosqldb import Row
from nosqldb import StoreConfig
from nosqldb import WriteOptions
##### Constants needed for the write options
from nosqldb import ONDB_DURABILITY
from nosqldb import ONDB_TIMEOUT

...
```

```
# locations where our store and proxy can be found
kvlite = 'localhost:5000'
proxy = 'localhost:7010'

...

# configure and open the store
def open_store():
    dg = Durability({ONDB_MASTER_SYNC : ONDB_SP_SYNC,
                     ONDB_REPLICA_SYNC : ONDB_SP_NO_SYNC,
                     ONDB_REPLICA_ACK : ONDB_AP_SIMPLE_MAJORITY})

    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvstoreconfig.set_durability(dg)
    return Factory.open(proxy, kvstoreconfig)
```

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:

```
def do_store_ops(store):
    row_d = { 'item' : 'bolts',
              'description' : "Hex head, stainless",
              'count' : 5,
              'percentage' : 0.2173913}
    row = Row(row_d)

    ## Create the write options

    dur = Durability({ONDB_MASTER_SYNC : ONDB_SP_NO_SYNC,
                      ONDB_REPLICA_SYNC : ONDB_SP_NO_SYNC,
                      ONDB_REPLICA_ACK : ONDB_AP_NONE})

    wo = WriteOptions({ONDB_DURABILITY : dur,
                       ONDB_TIMEOUT : 600})
    try:
        store.put("myTable", row, wo)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
    except DurabilityException, de:
        logging.error("Could not write table. Durability failure.")
        logging.error(de.message)
    except RequestTimeoutException, rte:
        logging.error("Could not write table. Exceeded timeout.")
        logging.error(rte.message)
```

# 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 the `Operation` class and `Store.execute_updates()` 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 error 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 `ONDB_ABORT_IF_UNSUCCESSFUL` for the `Operation` object then an `OperationExecutionException` is thrown. This error contains information about the failed operation.

## Creating a Sequence

You create a sequence by constructing an array of `Operation` objects. For each object, you specify the necessary operation information using an `OperationType` object provided to the object's `ONDB_OPERATION` key. Each element in the array represents exactly one operation in the store.

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:

```
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import Operation
from nosqldb import OperationType
from nosqldb import ProxyConfig
from nosqldb import Row
from nosqldb import StoreConfig

import logging
import os
import sys

### Constants needed for operations
from nosqldb import ONDB_OPERATION
from nosqldb import ONDB_OPERATION_TYPE
from nosqldb import ONDB_TABLE_NAME
from nosqldb import ONDB_ROW
from nosqldb import ONDB_ABORT_IF_UNSUCCESSFUL

op_array = []

...
## Skipped setup and logging functions for brevity
...

def add_op(op_t, table_name, if_unsuccess,
           item_type, item_cat, item_class,
           item_color, item_size, price, inv_count):

    global op_array

    row_d  = { 'itemType' : item_type,
               'itemCategory' : item_cat,
               'itemClass' : item_class,
               'itemColor' : item_color,
               'itemSize'  : item_size,
               'price'     : price,
               'inventoryCount' : inv_count
               }
    op_row = Row(row_d)

    op_type = OperationType({ONDB_OPERATION_TYPE : op_t})
    op = Operation({
                    ONDB_OPERATION : op_type,
                    ONDB_TABLE_NAME : table_name,
                    ONDB_ROW : op_row,
                    ONDB_ABORT_IF_UNSUCCESSFUL : True
                   })
    op_array.append(op)

...

if __name__ == '__main__':

    ...

    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "small", 13.07, 107)
    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "medium", 14.07, 198)
    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "large", 15.07, 140)

...
```

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 `Store.execute_updates()` method:

```
...

def do_store_ops(store):

    try:
        store.execute_updates(op_array)
        logging.debug("Store write succeeded.")
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
        sys.exit(-1)

if __name__ == '__main__':

    ...

    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "small", 13.07, 107)
    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "medium", 14.07, 198)
    add_op('PUT', 'myTable', True,
            "Hats", "baseball", "longbill",
            "red", "large", 15.07, 140)

    do_store_ops(store)
```

Note that if any of the above errors 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.

`Store.execute_updates()` can optionally take a `WriteOptions` object. This object allows you to specify:

- The durability guarantee that you want to use for this sequence. If you want to use the default durability guarantee, do not set this key, or set it to `None`.
- 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 `ONDB_RETURN_CHOICE` value that indicates whether you want to return the value resulting from the operation, the row's version, both of these things, or nothing.

For an example of using `WriteOptions`, see Durability Guarantees (page 72).

# 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 14).

For overview information on primary and shard keys, see Primary and Shard Key Design (page 21).

For overview information on indexes, see Creating Indexes (page 18).

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 21).

  To define a shard key (optional), use the SHARD keyword in the primary key statement. For information on shard keys, see Shard Keys (page 22).

  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 86) 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 86) 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 86) 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 84) 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 53).

## 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. Proxy Server Reference

The proxy server command line options are:

```
nohup java -cp KVHOME/lib/kvclient.jar:kvproxy/lib/kvproxy.jar
oracle.kv.proxy.KVProxy -help
 -port <port-number> Port number of the proxy server. Default: 5010
 -store <store-name> Required KVStore name. No default.
 -helper-hosts <host:port,host:port,...>   Required list of KVStore
        hosts and ports (comma separated).
 -security <security-file-path>  Identifies the security file used
        to specify properties for login. Required for connecting to
        a secure store.
  -username <user>  Identifies the name of the user to login to the
        secured store. Required for connecting to a secure store.
  -read-zones <zone,zone,...>  List of read zone names.
  -max-active-requests <int> Maximum number of active requests towards
        the store.
  -node-limit-percent <int> Limit on the number of requests, as a
        percentage of the requested maximum active requests.
  -request-threshold-percent <int> Threshold for activating request
        limiting, as a percentage of the requested maximum active
        requests.
  -request-timeout <long> Configures the default request timeout in
        milliseconds.
  -socket-open-timeout <long> Configures the open timeout in
        milliseconds used when establishing sockets to the store.
  -socket-read-timeout <long> Configures the read timeout in
        milliseconds associated with the underlying sockets to the
        store.
  -max-iterator-results <long> A long representing the maximum
        number of results returned in one single iterator call.
        Default: 100
  -iterator-expiration <long>  Iterator expiration interval in
        milliseconds.
  -max-open-iterators <int>    Maximum concurrent opened iterators.
        Default: 10000
  -num-pool-threads <int>      Number of proxy threads. Default: 20
  -max-concurrent-requests <int>      The maximum number of
        concurrent requests per iterator. Default: <num_cpus * 2>
  -max-results-batches <int>      The maximum number of results
        batches that can be held in the proxy per iterator.
        Default: 0
  -help  Usage instructions.
  -version  Print KVProxy server version number.
  -verbose  Turn verbose flag on.
```

Always start the Oracle NoSQL Database store before starting the proxy server.

When connecting to a non-secured store, the following parameters are required:

- `-helper-hosts`
- `-port`
- `-store`

When connecting to a secured store, the following parameters are also required:

- `-security`
- `-username`

### Note

Drivers are able to start and stop the proxy server on the local host if properly configured. See Automatically Starting the Proxy Server (page 5) for details.

## Securing Oracle NoSQL Database Proxy Server

If configured properly, the proxy can access a secure installation of Oracle NoSQL Database. To do this, the `-username` and `-security` proxy options must be specified.

The following example describes how to add security to an Oracle NoSQL Database single node deployment. The example also shows how to initiate a connection to the Oracle NoSQL Database replication nodes.

To install Oracle NoSQL Database securely:

```
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar makebootconfig \
-root KVROOT -port 5000 \
-admin 5001 -host node01 -harange 5890,5900 \
-store-security configure -pwdmgr pwdfile -capacity 1
```

1. Run the `makebootconfig` utility with the required `-store-security` option to set up the basic store configuration with security:

2. In this example, `-store-security configure` is used, so the `security configuration` utility is run as part of the makebootconfig process and you are prompted for a password to use for your keystore file:

   ```
   Enter a password for the Java KeyStore:
   ```

3. Enter a password for your store and then reenter it for verification. In this case, the password file is used, and the `securityconfig` tool will automatically generate the following security related files:

   ```
   Enter a password for the Java KeyStore: ***********
   Re-enter the KeyStore password for verification: ***********
   Created files:
   security/client.trust
   ```

```
security/client.security
security/store.keys
security/store.trust
security/store.passwd
security/security.xml
```

### Note

In a multi-host store environment, the security directory and all files contained in it should be copied to each server that will host a Storage Node. For more information on multiple node deployments see the Oracle NoSQL Database Security Guide.

4. Start the Storage Node Agent (SNA):

   ```
   nohup java -Xmx256m -Xms256m \
   -jar KVHOME/lib/kvstore.jar start -root KVROOT&
   ```

   When a newly created store with a secure configuration is first started, there are no user definitions available against which to authenticate access. To reduce risk of unauthorized access, an admin will only allow you to connect to it from the host on which it is running. This security measure is not a complete safeguard against unauthorized access. It is important that you do not provide local access to machines running KVStore. In addition, you should perform steps 5, 6 and 7 soon after this step to minimize the time period in which the admin might be accessible without full authentication. For more information on maintaining a secure store see the *Oracle NoSQL Database Security Guide*.

5. Start `runadmin` in security mode on the KVStore server host (node01). To do this, use the following command:

   ```
   java -Xmx256m -Xms256m \
   -jar KVHOME/lib/kvstore.jar \
   runadmin -port 5000 -host node01 \
   -security KVROOT/security/client.security
   Logged in admin as anonymous
   ```

6. Use the `configure -name` command to specify the name of the KVStore that you want to configure:

   ```
   kv-> configure -name mystore
   Store configured: mystore
   ```

7. Configure the KVStore by deploying a Zone, a Storage Node, and an Admin Node. Then, create a Storage Node Pool. Finally, create and deploy a topology.

   ```
   kv-> plan deploy-zone -name mydc -rf 1 -wait
   Executed plan 2, waiting for completion...
   Plan 2 ended successfully
   kv-> plan deploy-sn -zn zn1 -port 5000 -host node01 -wait
   Executed plan 3, waiting for completion...
   Plan 3 ended successfully
   kv-> plan deploy-admin -sn sn1 -port 5001 -wait
   ```

```
Executed plan 4, waiting for completion...
Plan 4 ended successfully
kv-> pool create -name mypool
kv-> pool join -name mypool -sn sn1
Added Storage Node(s) [sn1] to pool mypool
kv-> topology create -name mytopo -pool mypool -partitions 30
Created: mytopo
kv-> plan deploy-topology -name mytopo -wait
Executed plan 5, waiting for completion...
Plan 5 ended successfully
```

8. Create an admin user. In this case, user `root` is defined:

```
kv-> plan create-user -name root -admin -wait
Enter the new password: ********
Re-enter the new password: ********
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

9. Create a new password file to store the credentials needed to allow clients to login as the admin user (root):

```
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar securityconfig \
pwdfile create -file KVROOT/security/login.passwd
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar securityconfig pwdfile secret \
-file KVROOT/security/login.passwd -set -alias root
Enter the secret value to store: ********
Re-enter the secret value for verification: ********
Secret created
OK
```

### Note

The password must match the one set for the admin in the previous step.

10. At this point, it is possible to connect to the store as the root user. To login, you can use either the `-username <user>` runadmin argument or specify the "oracle.kv.auth.username" property in the security file.

In this example, a security file (`mylogin.txt`) is used. To login, use the following command:

```
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar runadmin -port 5000 \
-host localhost -security mylogin
Logged in admin as root
```

The file `mylogin.txt` should be a copy of the `client.security` file with additional properties settings for authentication. The file would then contain content like this:

```
oracle.kv.auth.username=root
oracle.kv.auth.pwdfile.file=KVROOT/security/login.passwd
oracle.kv.transport=ssl
oracle.kv.ssl.trustStore=KVROOT/security/client.trust
oracle.kv.ssl.protocols=TLSv1.2,TLSv1.1,TLSv1
oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
```

Then, to run KVProxy and access the secure Oracle NoSQL Database deployment:

```
java -cp KVHOME/lib/kvclient.jar:KVPROXY/lib/kvproxy.jar
oracle.kv.proxy.KVProxy -helper-hosts node01:5000 -port 5010
-store mystore -username root -security mylogin
Nov 21, 2014 12:59:12 AM oracle.kv.proxy.KVProxy <init>
INFO: PS: Starting KVProxy server
Nov 21, 2014 12:59:12 AM oracle.kv.proxy.KVProxy <init>
INFO: PS: Connect to Oracle NoSQL Database mystore nodes : localhost:5000
Nov 21, 2014 12:59:13 AM oracle.kv.proxy.KVProxy <init>
INFO: PS:   ... connected successfully
Nov 21, 2014 12:59:13 AM oracle.kv.proxy.KVProxy startServer
INFO: PS: Starting listener ( Half-Sync/Half-Async server - 20
no of threads on port 5010)
```

### Note

Because this proxy server is being used with a secure store, you should limit the proxy server's listening port (port 5010 in the previous example) to only those hosts running authorized clients.

## Trouble Shooting the Proxy Server

If your client is having trouble connecting to the store, then the problem can possibly be with your client code, with the proxy and its configuration, or with the store. To help determine what might be going wrong, it is useful to have a high level understanding of what happens when your client code is connecting to a store.

1. First, your client code tries to connect to the `ip:port` pair given for the proxy.

2. If the connection attempt is not successful, and your client code indicates that the proxy should be automatically started, then:

   a. The client driver will prepare a command line that starts the proxy on the local host. This command line includes the path to the `java` command, the classpath to the two jar files required to start the proxy, and the parameters required to start the proxy and connect to the store (these include the local port for the proxy to listen on, and the store's connection information).

   b. The driver executes the command line. If there is a problem, the driver might be able to provide some relevant error information, depending on the exact nature of the problem.

   c. Upon command execution, the driver waits for a few seconds for the connection to complete. During this time, the proxy will attempt to start. At this point it might indicate a problem with the classpath.

      Next, it will check the version of `kvclient.jar` and indicate if it is not suited.

      After that, it will check the connection parameters, and indicate problems with those, if any.

      Then the proxy will actually connect to the store, using the `helper-hosts` parameter. At this time, it could report connection errors such as the store is not available, security credentials are not available, or security credentials are incorrect.

      Finally, the proxy tries to listen to the indicated port. If there's an error listening to the port (it is already in use by another process, for example), the proxy reports that.

   d. If any errors occur in the previous step, the driver will automatically repeat the entire process again. It will continue to repeat this process until it either successfully obtains a connection, or it runs out of retry attempts.

      Ultimately, if the driver cannot successfully create a connection, the driver will return with an error.

3. If the driver successfully connects to the proxy, it sends a verify message to the proxy. This verify message includes the helper-host list, the store name, the username (if using a secure store), and the readzones if they are being used in the store.

   If there is anything wrong with the information in the verify message, the proxy will return an error message. This causes the proxy to check the verify parameters so as to ensure that the driver is connected to the right store.

4. If there are no errors seen in the verify message, then the connection is established and store operations can be performed.

To obtain the best error information possible when attempting to troubleshoot a connection problem, start the proxy with the `-verbose` command line option. Also, you can enable assertions in the proxy Java code by using the `java -ea` command line option.

Between these two mechanisms, the proxy will provide a great deal of information. To help you analyze it, you can enable logging to a file. To do this:

Start the proxy with the following parameter:

```
java -cp KVHOME/lib/kvclient.jar:KVPROXY/lib/kvproxy.jar
-Djava.util.logging.config.file=logger.properties
oracle.kv.proxy.KVProxy -helper-hosts node01:5000 -port 5010
-store mystore -verbose
```

The file `logger.properties` would then contain content like this:

```
# Log to file and console
```

```
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
## ConsoleHandler ##
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter =
                                       java.util.logging.SimpleFormatter
## FileHandler ##
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
# Limit the size of the file to x bytes
java.util.logging.FileHandler.limit = 100000
# Number of log files to rotate
java.util.logging.FileHandler.count = 1
# Location and log file name
# %g is the generation number to distinguish rotated logs
java.util.logging.FileHandler.pattern = ./kvproxy.%g.log
```

Configuration parameters control the size and number of rotating log files used (similar to java logging, see java.util.logging.FileHandler). For a rotating set of files, as each file reaches a given size limit, it is closed, rotated out, and a new file is opened. Successively older files are named by adding "0", "1", "2", etc. into the file name.

# Appendix C. Third Party Licenses

All of the third party licenses used by the Oracle NoSQL Database Python driver are described in the `LICENSE` file, which you can find in the `nosqldb` directory, which exists where you install your Python modules.