Oracle® NoSQL Database

Python Driver Developer's Guide

Release 20.1

ORACLE®

Contents

4   Primary and Shard Key Design

5   Writing and Deleting Table Rows

6   Reading Table Rows

7   Using Data Types

8   Indexing Non-Scalar Data Types

9   Using Row Versions

10   Consistency Guarantees

11   Durability Guarantees

12   Executing a Sequence of Operations

A   Table Data Definition Language Overview

B   Proxy Server Reference

C   Third Party Licenses

Preface

There are two different APIs that can be used to write 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), Python, and C#. 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 not found in the Key/Value API. The Key/Value API will no longer be enhanced in future releases of Oracle NoSQL Database.

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.

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
6. C# Table Driver

> **Note:**
>
> New users should use one of the Table drivers unless they require a feature only available in the Key/Value API (such as Large Object support). The Key/Value API will no longer be enhanced in future releases of Oracle NoSQL Database.

The Java and C Key/Value driver provides access to store data using key/value pairs. All other drivers provide access using tables. Also, 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 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, node.js Table, and C# 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:**
>
> The Python 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 here.

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. The host names provided here must be resolvable using either DNS or the local `/etc/hosts` file.

- `-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.

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.

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. See Setting the Security Properties for a Proxy Server.

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 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 .

```
# 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.

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. See Consistency Guarantees.

- 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. See Durability Guarantees.

- set_max_results()

  The number of rows buffered by iterators.

- set_read_zones()

  An array of zone names to be used as read zones. See *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 most common properties you will set are listed here. For the complete list, see *Python Driver API Reference*.

- 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.

- 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

You can install Oracle NoSQL Database so 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 securely, you can require authentication. Requiring authentication incurs a performance cost, due to the overhead of using SSL and authentication. While we recommend that your production store requires authentication over SSL, some sites that are particularly performance sensitive can 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.

If you configure your store to support authentication, it is automatically configured to communicate with clients using SSL. The use of SSL ensures privacy of the authentication and other sensitive information. To use SSL, you must install SSL certificates on the machines where your proxy server runs, to validate that the store being accessed is trustworthy.

For information on configuring a store for authentication, see Configuring Authentication in the *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 Security Configuration in the *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 for an example of how this is done.

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.

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.
- Security is turned on.

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

For example:

```
> java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar kvlite
```

> **Note:**
>
> To avoid using too much heap space, you should specify the `-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 (the following assumes security is enabled):

```
Generated password for user admin: password
User login file: ./kvroot/security/user.security
Created new kvlite store with args:
```

```
-root ./kvroot -store <kvstore> -host localhost -port 5000
-secure-config enable
```

> **Note:**
>
> The password is randomly generated.

or

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

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 -Xmx64m -Xms64m -jar lib/kvclient.jar

This should write the release to stdout:

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

- Download the examples package and unpack it so that the examples directory is in KVHOME. You can obtain the examples package from the same place as you obtained your server download package.

- 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 -Xmx64m -Xms64m \
     -Doracle.kv.security=<KVROOT>/security/user.security \
     -cp lib/kvclient.jar:examples hello.HelloBigDataWorld
     ```

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

     ```
     java -Xmx64m -Xms64m \
     -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.

- `-help`

  Print a brief usage message, and exit.

- `-host <hostname>`

  Identifies the name of the host on which KVLite is running.

  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.

- `-noadmin`

  If this option is not specified, the administration user interface is started.

- `-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.

- `-secure-config <enable|disable>`

  If enabled, causes security to be enabled for the store. This means all clients connecting to the store must present security credentials. Security is enabled by default.

- `-store <storename>`

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

  For information on configuring your proxy server to connect to a secure store, see Working with a Secured Store .

3

Introducing Oracle NoSQL Database Tables and Indexes

Using the Table API (in one of the supported languages) is the recommended method of developing an Oracle NoSQL Database client application. Table APIs let you manipulate data using a tables metaphor, in which data is organized in multiple columns of data. The table APIs support an unlimited number of subtables. You can also create indexes to improve query performance against your tables.

If you have a mix of clients accessing your store using both Table and Key/Value APIs, a remote chance exists that keys from different clients could collide. To avoid any possible conflict between keys, however unlikely, make sure that every KV key has either only:

- A single component
- A single major component

Defining Tables

Before an Oracle NoSQL Database client can read or write to a table in the store, you must first create the tables. There are several ways to do this, but this document focuses on using Table DDL statements. You can submit these statements to the store directly using both the Admin command line interface (CLI), with the `execute` command), or the SQL CLI. However, the recommended approach is to submit DDL statements to the store programmatically. This section describes both direct and programmatic methods.

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

As an introductory example, suppose you want to create a table called `myTable` with four columns: `item`, `description`, `count`, and `percentage`. To create your table, use the following statement from the SQL CLI:

```
sql-> CREATE TABLE myTable (
    item STRING,
    description STRING,
    count INTEGER,
    percentage DOUBLE,
    PRIMARY KEY (item) // Every table must have a primary key
);
Statement completed successfully
```

> **✎ Note:**
>
> Primary keys are a concept that have not yet been introduced. See Primary and Shard Key Design for a complete explanation of what they are, and how you should use them. For now, be sure a primary key exists for every table you create, just as the previous example illustrates.

Executing DDL Statements Programmatically

To add the table definition to the store programmatically use the `Store.execute()` or `Store.execute_sync()` methods. (The latter method executes the statement synchronously.)

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 From the Admin CLI

You can execute DDL statements using the Admin 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.

Oracle NoSQL Database tables support the following data types:

Data Type	Description
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	Implemented as a byte array with no predetermined fixed size.
Float	
Integer	
Json	Any valid JSON data.
Long	
Number	A numeric type capable of handling any type of number or any value or precision.
Map	An unordered map type, where all entries are constrained by a single type.
Records	See the following section.
String	
Timestamp	An absolute timestamp encapsulating a date and, optionally, a time value.

For Python, supported data types are handled as follows:

Oracle NoSQL Database Datatype	Python Datatype
Array	Python array. See Using Arrays.
Binary	Base64 encoded buffer. See Using Binary.
Boolean	Python boolean
Double	Python long
Enum	Python string. See Using Enums.
Fixed Binary	Base64 encoded buffer. See Using Fixed Binary.
Float	Python float
Integer	Python int
Long	Python long
Map	Python dictionary. See Using Maps.
Records	Python dictionary. See Using Embedded Records.
String	Python string

Record Fields

As described in Defining Child Tables, 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, you do not need to create child tables to organize subordinate data. If you have simple requirements for subordinate data, you can use record fields, instead of child tables. In general, you can use record fields instead of child tables if you want only a fixed, small number of instances of the record for each parent table row. For anything beyond trivial cases, use child tables.

> ✎ **Note:**
>
> There is no downside to using child tables for even trivial cases.

The assumption when using record fields is that you have a fixed, known number of records to manage (unless you organize them as arrays). For example, in a contacts database, child tables let you have an unlimited number of addresses associated for each user. By using records, rather than child tables, 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),
    workAddress RECORD (street STRING, city STRING, state STRING,
               zip INTEGER),
    PRIMARY KEY(uid))
```

Alternatively, you can create an array of record fields. This lets you create an unlimited number of address records per field. In general, however, 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))),
    PRIMARY KEY(uid))
```

Defining Tables With an IDENTITY Column

You can create an IDENTITY column to auto-increment a value each time you add a row.

You create an IDENTITY column as part of a `CREATE TABLE` *name* DDL statement, or add an IDENTITY column to an existing table with an `ALTER TABLE` *name* DDL statement.

Only one IDENTITY column can exist per table. It must be an `INTEGER`, `LONG`, or `NUMBER` datatype.

Every IDENTITY column is part of a table, and cannot exist as an independent object. For more information about adding IDENTITY columns, see Altering or Dropping an IDENTITY Column in the *Java Direct Driver Developer's Guide*.

An IDENTITY column requires an associated Sequence Generator (SG). The SG is the table's manager for tracking the IDENTITY column's current, next, and total number of values. An SG has several attributes that define its behavior, such as the starting value for its IDENTITY column, or the number of values stored in cache. You can optionally define some SG attributes when you create an IDENTITY column, or use all default values. For more information about the Sequence Generator attributes, see Sequence Generator Attributes in the *Java Direct Driver Developer's Guide*.

> **✎ Note:**
>
> Using an IDENTITY column in any table does not force uniqueness. If your application requires unique values for every row of an IDENTITY column, you must create the column as GENERATED ALWAYS AS IDENTITY, and never permit any use of the CYCLE SG attribute.

If more than one client accesses a table with an IDENTITY column defined for unique values this way, each client is assigned contiguous value sets to its SG cache. These sets do not overlap with other client sets. For example, `Client1` is assigned values `0001 – 1000`, while `Client2` has `1001 – 2000`, and so on. Thus, as each client adds rows to the table, the IDENTITY values can run as 0001, 1001, 0002, 1002, 1003, and so on, as both clients use their own cache when adding rows. The IDENTITY column values are guaranteed to be unique, but not necessarily contiguous, because each client has its own set of cache values, and adds rows at different speeds and times.

Users require table privileges to create tables with an IDENTITY column. For a description of user privileges, see KVStore Required Privileges in the *Security Guide*.

Creating Tables With an IDENTITY Column

You can create an IDENTITY column when you create a table, or change an existing table to add an IDENTITY column using ALTER TABLE...ADD. In either case, choose

one of the IDENTITY statements described below. This section describes creating a table with an IDENTITY column.

Here is the formal syntax for creating a table with an IDENTITY column:

```
GENERATED (ALWAYS | (BY DEFAULT [ON NULL])) AS IDENTITY
   [sequence_options,...]
```

The optional `sequence_options` refer to all of the Sequence Generator attributes you can supply.

IDENTITY Column Statement	Description
GENERATED ALWAYS AS IDENTITY	The sequence generator always supplies an IDENTITY value. You cannot specify a value for the column.
GENERATED BY DEFAULT AS IDENTITY	The sequence generator supplies an IDENTITY value any time you do not supply a column value.
GENERATED BY DEFAULT ON NULL AS IDENTITY	The sequence generator supplies the next IDENTITY value if you specify a NULL columnn value.

To create a table with a column GENERATED ALWAYS AS IDENTITY from the SQL CLI:

```
sql-> CREATE TABLE IF NOT EXISTS tname1 (
idValue INTEGER GENERATED ALWAYS AS IDENTITY,
acctNumber INTEGER,
name STRING,
PRIMARY KEY (acctNumber));
Statement completed successfully
sql->
```

For this table, `tname1`, each time you add a row to the table, the Sequence Generator (SG) updates the `idvalue` from its cache. You cannot specify a value for `idValue`. If you do not specify any sequence generator attributes, the SG uses its default values.

To create a table with a column GENERATED BY DEFAULT ON NULL AS IDENTITY:

```
sql-> CREATE TABLE IF NOT EXISTS tname2 (
idvalue INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
acctNumber INTEGER,
name STRING,
PRIMARY KEY (acctNumber));
Statement completed successfully
sql->
```

For this table, `tname2`, each time you add a row, the SG inserts the next available value from its cache if no value is supplied for the `idvalue` column, the supplied value for the `idvalue` column is NULL.

To create a table with a column GENERATED BY DEFAULT AS IDENTITY:

```
sql-> CREATE TABLE IF NOT EXISTS tname3 (
idvalue INTEGER GENERATED BY DEFAULT AS IDENTITY,
acctNumber INTEGER,
name STRING,
PRIMARY KEY (acctNumber));
Statement completed successfully
sql->
```

For this table, `tname3`, each time you add a row, the SG inserts the next available value from its cache if no value is supplied for the `idvalue` column.

To create a new table, `sg_atts`, with several SG attributes:

```
sql-> CREATE Table sg_atts (
id INTEGER GENERATED ALWAYS AS IDENTITY
(START WITH 2
INCREMENT BY 2
MAXVALUE 200
NO CYCLE),
name STRING,
PRIMARY KEY (id));
Statement completed successfully
sql->
```

The table `sg_atts` specifies that the integer IDENTITY field (`id`) is generated always.

SG Attribute	Description
start with 2	Start the sequence value at 2.
increment by 2	Increment the sequence value by 2 for each row.
maxvalue 200	Specifies the maximum IDENTITY value. What you specify overrides the default value maxvalue, which is the upper bound of the IDENTITY datatype in use. Once the IDENTITY column reaches this value, 200, the SG will not generate any more IDENTITY values. The maximum value has been reached and the no cycle attribute is in use.
no cycle	Do not restart from 2 or with any value at all, once the column reaches the `maxvalue`.

To create another table, `sg_some_atts`, with some SG attributes:

```
sql-> CREATE Table sg_some_atts (
id LONG GENERATED BY DEFAULT AS IDENTITY
(START WITH 1
INCREMENT BY 1
CYCLE
CACHE 200),
account_id INTEGER,
name STRING,
PRIMARY KEY (account_id));
```

For the `sg_some_atts` table, specify an `id` column `GENERATED BY DEFAULT AS IDENTITY`, but which is not the primary key.

SG Attribute or Other Detail	Description
`CYCLE`	Specifying CYCLE indicates that the SG should supply IDENTITY values up to either the `MAXVALUE` attribute you specify, or the default `MAXVALUE`. When the IDENTITY reaches the `MAXVALUE` value, the SG restarts the values over, beginning with `MINVALUE`, if it is specified, or with the default `MINVALUE` for the data type. CYCLE is orthogonal to the CACHE attribute, which indicates only how many values to store in local cache for swift access. You can set CACHE value to closely reflect the maximum value of the datatype, but we do not recommend this, due to the client cache size.
`CACHE 200`	The number of values that each client stores in its cache for fast retrieval. When the IDENTITY reaches the last number in the cache, the SG gets another set of values from the server automatically.
`START WITH 1`	The SG generates values 1, 2, 3 and so on, until it reaches the maximum value for a LONG data type.
`INCREMENT BY 1`	The SG increments each new IDENTITY value for every new row.

For a full list of all sequence generator attributes, see Sequence Generator Attributes in the *Java Direct Driver Developer's Guide*.

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 for more information.

Defining Multi-Region Tables

A Multi-Region Table or MR Table is a global logical table that is stored and maintained in different regions or installations. It is a *read-anywhere* and *write-anywhere* table that lives in multiple regions.

Consider an Oracle NoSQL Database with three regions, Frankfurt, London, and Dublin. To create a table called `Users` that stores user details for all the three regions, you must create an MR table on each KVStore in the connected graph, and specify the list of regions that the table should span.

For example, to create the `Users` table in the Frankfurt region specifying London and Dublin as its remote regions, you must execute the following code from the Frankfurt region:

```
CREATE TABLE Users (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  primary key (id)
) IN REGIONS London, Dublin;
```

To create the `Users` table as an MR table in all the three regions, you must execute the `create table` statement in the London and Dublin regions also, specifying the other two regions as its remote regions.

> ✎ **Note:**
>
> TTL feature is not supported for MR Tables in this preview release. This is a temporary limitation that will be removed in a future release.

For information about MR Tables, see Life Cycle of MR Tables in the *Concepts Guide*.

Table Evolution

As your application is used over time, it's often necessary to update your tables to either add new fields or remove existing fields that are no longer required. Table evolution is the term used to update table definitions, adding or removing fields, or changing field properties, such as a default value. You may even add a particular kind of column, like an IDENTITY column, to increment some value automatically. Only tables that already exist in the store are candidates for table evolution.

Use the `ALTER TABLE` statement to perform table evolution. See Modify Table Definitions.

> **Note:**
>
> You cannot remove a field if it is a primary key field, or if it participates in an index. You also cannot add primary key fields during table evolution.

For example, the following statements evolve the table that was created in the previous section. In this example, you would submit each statement to the store consecutively, 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.

You can create indexes on any field that has a data type capable of indexing, including primary key fields. You can index table IDENTITY fields. For information on the types of fields that can be indexed, see Indexable Field Types.

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 red automobiles, regardless of the manufacturer or model type, you could create an index on the field containing 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 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. For example:

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

Similarly, to remove an index, use the `DROP INDEX` statement. See DROP INDEX. For example:

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

Be aware that adding and dropping indexes can be time consuming. You may want to run drop index 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)
```

For examples of how to index supported non-scalar types, see Indexing Non-Scalar Data Types.

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. We show how to delete multiple rows at a time in Using multi_delete() .

> **Note:**
>
> If the primary key field is an INTEGER data type, you can apply a serialized size constraint to it. See Integer Serialized Constraints.

Data Type Limitations

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

- Integer
- Long
- Number
- Float
- Double
- String
- Timestamp
- 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 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.

5

Writing and Deleting Table Rows

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

Write Exceptions

There are many exceptions that you should handle whenever you perform a write operation to the store. Some of the more common exceptions are described here. For simple cases where you use default policies or are not using a secure store, you can probably avoid explicitly handling these. However, as your code complexity increases, so too will the desirability of explicitly managing these exceptions.

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

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

Writing Rows to a Table in the Store

Writing a new row to a table in the data store, and updating an existing row are similar operations. Later in this section, we describe methods that work only if a row is being updated, or only if you are creating a row. You can write data to a table only after it has been added to the store. See Introducing Oracle NoSQL Database Tables and Indexes for details.

To write a row to a table in the store:

Writing Rows to a Child Table

To write to a child table, complete the tasks that you do for a parent table, except using the two-part table name, such as *parent-table.child-table*.

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

Writing Rows to a Table with an IDENTITY Column

After creating a table with an IDENTITY column, you can write new rows into it programmatically.

You must create a table with an IDENTITY column before writing data into it. For more information, see Creating Tables With an IDENTITY Column. The example below explains how to create a table with an IDENTITY column and write data into it using the Python driver code.

1. Create a table called `myTable` with an IDENTITY column by executing the below command on the SQL CLI.

    ```
    sql-> CREATE TABLE myTable (
    id INTEGER GENERATED ALWAYS AS IDENTITY (
    START WITH 1 INCREMENT BY 1 MAXVALUE 10 CYCLE CACHE 3),
    name STRING,
    PRIMARY KEY (id));
    Statement completed successfully
    sql->
    ```

2. To write data to the `myTable` table, execute the below python code:

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

    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 = Row({'name':'joe'})
    try:

        store.put("myTable", row_d)
        identity_value = row_d.get_identity_value()
        logging.debug("Store write succeeded.")

    except IllegalArgumentException, iae:

        logging.error("Could not write table.")
        logging.error(iae.message)
    ```

```
store.close()
```

> **Note:**
>
> Make sure that the `kvclient` and `kvproxy` are up and running before executing this code.

> **Tip:**
>
> Place the above code into `.py` file and execute it from the Python Shell or command line interface.

3. Execute the below command on the SQL CLI to verify the row insertion:

```
sql-> select * from myTable;
{"id":1,"name":"joe"}

1 row returned
```

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 .

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 .

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

```
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, as long as they all share the shard key values. Recall that shard keys are at least a subset of your primary keys. This results in using a partial primary key, which is the shard key, to perform a multi-delete.

To delete multiple rows at once, use the `Store.multi_delete()` method.

For example, suppose you create 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

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)
```

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 set of keys as the retrieval criteria. Use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide bulk retrievals.

Each of these are described in the following sections.

Read Exceptions

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

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

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

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

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

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

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

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 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 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.

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
                            shard_key_d, # partial primary key
                            False)       # Retrieve only keys?
        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.

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.

```python
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)
```

```python
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)
```

Bulk Get Operations

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

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

To use bulk get, use one of the methods that provide bulk retrievals. These accept a set of keys instead of a single key as the retrieval criteria. The set is provided using either an `Iterator<Key>` or `List<Iterator<Key>>` value.

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

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

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)
)
```

`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.

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 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)
)
```

`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)
)
```

DEFAULT and NOT NULL constraints are supported for enumerated fields. See DEFAULT 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.

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 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)
)
```

`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 JSON

The JSON datatype is available as a preview feature.

The JSON datatype cannot be used as part of a primary or shard key. In this release, it also can not participate in an index.

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

```
CREATE TABLE myJsonTable (
    uid INTEGER,
    myJSON JSON,
    PRIMARY KEY (uid)
)
```

The data that you write for this datatype can be any valid JSON stored as a string. For example, all of the following are valid:

```
jsonNumber = "2"
jsonString = "\"a json string\""
jsonObject_null = "{}"
jsonObject = '{"a": 1.006, "b": null, \
  "bool" : true, "map": {"m1": 5}, \
  "ar" : [1,2.7,3]}'
String jsonNull = "null"
```

To store a JSON value in the table that we defined, above:

```
row_d = {'uid' : 0,
         'myJSON' : '[1,2.8,1001,322987]'
        }
try:
    store.put("myJsonTable", row_d)
    logging.debug("Store write succeeded.")
except IllegalArgumentException, iae:
    logging.error("Could not write table.")
    logging.error(iae.message)
    sys.exit(-1)
```

> **✎ Note:**
>
> To store a JSON object, write it as a dictionary. For example:
>
> ```
> row_d = {'uid' : 0,
>          'myJSON' : {"field_one" : "a name",
>                      "field_two" : 10}}
> ```
>
> After that, you can write it to the store in the same way as the previous example shows.

To retrieve it:

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tuid: %s" % row['uid']
            print "\tJSON: %s" % row['myJSON']
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

...

def read_store(store):
    key_d = {}
    try:
        row_list = store.table_iterator("myJsonTable", 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)
```

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)
)
```

`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)
)
```

`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 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)
```

8

Indexing Non-Scalar Data Types

We describe how to index scalar data types in Creating Indexes, and we show how to read using indexes in Reading Indexes. 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 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. Be sure to use `[]` with the field name to indicate that it is an array:

```
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 JSON Fields

You can create an index on a JSON field. To create the index, specify it as you would any other index, except that you must define the data type of the JSON field you are indexing.

Note that there are some restrictions on the data type of the JSON field that you can index. See JSON Indexes for more information.

To create the index, first create the table:

```
CREATE Table JSONPersons (
    id INTEGER,
    person JSON,
    PRIMARY KEY (id)
)
```

To create the index, you must specify the JSON field to be indexed using dot notation. Suppose your table rows look like this:

```
"id":1,
"person" : {
    "firstname":"David",
```

```
        "lastname":"Morrison",
        "age":25,
        "income":100000,
        "lastLogin" : "2016-10-29T18:43:59.8319",
        "address":{"street":"150 Route 2",
                    "city":"Antioch",
                    "state":"TN",
                    "zipcode" : 37013,
                    "phones":[{"type":"home", "areacode":423,
                                "number":8634379}]
                },
        "connections":[2, 3],
        "expenses":{"food":1000, "gas":180}
   }
```

Then once the table has been added to the store, you can create an index for one of the JSON fields like this:

```
CREATE INDEX idx_json_income on JSONPersons (person.income AS integer)
```

To retrieve data using a JSON index, construct an index key using dot notation to identify the JSON field. You then use `Store.index_iterator()` to access the table rows that match the index key. For example, to retrieve all rows where the `person.income` field is 100000:

```
def display_row(row):
    try:
            print "Retrieved row:"
            print "\tid: %s" % row['id']
            print "\tperson: %s" % row['person']
    except KeyError, ke:
        logging.error("Row display failed. Bad key: %s" % ke.message)

def read_store(store):
    store.refresh_tables()

    try:
        key_d = {"person.income" : 100000}
        row_list = store.index_iterator("JSONPersons",
                    "idx_json_income",
                    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)
```

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.

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 `.keys()` path step:

```
CREATE INDEX mapKeyIndex on myMapTable (testMap.keys())
```

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.

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 `.values()` path step:

```
CREATE INDEX mapElementIndex on myMapTable (testMap.values())
```

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 the `.keys()` and `.values()` path steps:

```
CREATE INDEX mapKeyValueIndex on myMapTable
(testMap.keys(),testMap.values())
```

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)
```

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 perform the operation only 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.

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:

```python
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)
```

10

Consistency Guarantees

A Oracle NoSQL Database 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 nature of distributed systems, there is 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. The value returned is either equal or newer than the version specified in the policy.

- `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()
```

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. For more information on zones, see *Administrator's Guide*.

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.

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 not wait for acknowledgments, or to wait for acknowledgments from a simple majority of replica nodes in primary zones, or to wait for 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.

See the next section for an example of setting durability policies.

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 (> 50%) 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)
```

ORACLE

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 exception that is considered a fault. For example, the operation throws a `DurabilityException`. Also, if there is an internal failure due to message delivery or a networking error.

2. An individual operation returns normally but is unsuccessful as defined by the particular operation. (For example, you attempt to delete a row that does not exist). If this occurs AND you specified `true` for `ONDB_ABORT_IF_UNSUCCESSFUL` for the `Operation` object then an `OperationExecutionException` is thrown. This exception 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:

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

`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.

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. The best way to run DDL statements is from the SQL Shell as described in Running the SQL Shell in the *SQL Beginner's Guide*.

From the SQL, you can use these statements:

- Define tables and sub-tables.
- Modify table definitions.
- Delete table definitions.
- Define indexes.
- Delete index definitions.
- Set and modify default Time-to-Live values for tables.

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:

```
SQL-> CREATE TABLE Users (
    id INTEGER,
    firstName STRING,
    lastName STRING,
    contactInfo JSON,
    PRIMARY KEY (id)
)
```

For information on how to transmit these statements to the store, see Introducing Oracle NoSQL Database Tables and Indexes.

For overview information on primary and shard keys, see Primary and Shard Key Design.

For overview information on indexes, see Creating Indexes.

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

Throughout this document, and elsewhere in the documentation, using uppercase text signifies DDL keywords (such as STRING, CREATE TABLE, and so on). These keywords are actually case-insensitive and you can enter them in lowercase

characters. However, all DDL keywords shown here are reserved words. You cannot use keywords as table, index, or field names.

Table, index, and field names are case-preserving, but case-insensitive. For example, you can create a field named `MY_NAME`, and later reference it as `my_name` without error. However, whenever you display the field name, it displays in the way you created it, `MY_NAME`, in this case.

Table names are limited to 256 characters. Namespace names, index names, and field names are limited to 64 characters. All table, index, and field names must begin with a letter (A-Z, a-z), and are restricted to alphanumeric characters (A-Z, a-z, 0–9), plus underscore (_) and a period (.) character.

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, as follows:

```
CREATE TABLE [IF NOT EXISTS] [namespace:]table-name
 [COMMENT "comment string"]
 (field-definition, field-definition-2 [,...]
    PRIMARY KEY (field-name, field-name-2 [,...] ),
) [USING TTL ttl]
  [IN REGIONS region-name,region-name-2 [,...]]
```

where:

*   `IF NOT EXISTS`

    Optional clause. If you use this clause and a table of the same name and definition already exists in the current namespace, the statement neither creates a new table, nor returns an error. No action occurs.

    If you do not use `IF NOT EXISTS`, and a table of the same name and definition already exists in the current namespace, the statement attempts to create the table, and fails. You cannot have two tables with the same name in one namespace.

*   `table-name`

    Required. Specifies the full table name, regardless of where it exists in the table hierarchy. A table can be a top-level table created in the default namespace (`sysdefault`), a table in a non-default namespace, or a child or grandchild table of a parent table. Specify a fully-qualified table name as follows:

    –   `parent-table.child-table` – Specifies a new child table of an existing parent. To create a child table, specify the parent table followed by a period (.) before

the child name. For example, if the parent table is `Users`, define a child table named `MailingAddress` as `Users.MailingAddress`.

- *parent-table.child-table.grandchild-table* – Specifies a child table of an existing table. You must also specify the parent table, for example `Users.MailingAddress.Zip`.

- *namespace-name:table-name* – Identifies a new table in a specific namespace, either user-created or in the default namespace (`sysdefault:`). Use the namespace followed by a colon (:) as a prefix to the new table name. For a new child table of a table in a namespace, or any further generation, use a fully qualified table name such as `Sales:Users.MailingAddress`, or `Sales:Users.MailingAddress.Zip`.

- COMMENT

  Optional. Use a comment to provide a brief description of the table. The comment is not interpreted at runtime, but becomes part of the table's metadata.

- *field-definition*

  Required. A comma-separated list of fields. One of the columns can be an IDENTITY, as described in Defining Tables With an IDENTITY Column. There are one or more field definitions for every table. Field definitions are described next in this section.

- PRIMARY KEY

  Required for every table. Identifies at least one field in the table that is the primary key. For information on primary keys, see Primary Keys.

  > **✎ Note:**
  >
  > If the primary key field is an INTEGER data type, you can apply a serialized size constraint to it. See Integer Serialized Constraints for details.

  To optionally define a shard key, use the `SHARD` keyword within the primary key statement. For information on shard keys, see Shard Keys.

  For example:

  ```
  PRIMARY KEY (SHARD(id), lastName)
  ```

- USING TTL

  Optional. Defines a default time-to-live value for the table's rows. See USING TTL for information on this clause.

  > **✎ Note:**
  >
  > As of the latest release, the TTL feature is not supported for MR Tables in this preview release. This is a temporary limitation that will be removed in a future release.

- IN REGIONS

> Optional. In case, the table being created is an MR Table, this parameter lists all the regions that the table should span. You must mention at least one remote region in this clause to create the table as an MR Table. For information on MR Tables, see Life Cycle of MR Tables in the *Concepts Guide*.

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, allowable ranges or default values. Ann IDENTITY field, to be created by a sequence generator, is also permissable. This information is optional. See Field Constraints 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)
  ```

- `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.

- `JSON`

  A JSON-formatted string.

- `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)
```

- Number

  A numeric type capable of handling any type of number of any value or precision.

- `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.

- `TIMESTAMP(<precision>)`

  Represents a point in time as a date and, optionally, a time value.

  Timestamp values have a precision (0 - 9) which represents the fractional seconds to be held by the timestamp. A value of 0 means that no fractional seconds are stored, 3 means that the timestamp stores milliseconds, and 9 means a precision of nanoseconds. When declaring a timestamp field, the precision is required.

Field Constraints

Field constraints define information about the field, such as whether the field can be NULL, or what a row's default value should be. Not all data types support constraints, and individual data types do not support all possible constraints.

Integer Serialized Constraints

You can put a serialized size constraint on an INTEGER data type, provided the INTEGER is used for a primary key field. Doing this can reduce the size the keys in your store.

To do this, use `(n)` after the primary key field name, where *n* is the number of bytes allowed for the integer. The meaningful range for *n* is 1 - 4. For example:

```
create table myId (id integer, primary key(id(3)))
```

The number of bytes allowed defines how large the integer can be. The range is from negative to positive.

> **Note:**
>
> Specifying an integer constraint value for number of bytes on an IDENTITY field is not permitted.

Number of Bytes	Allowed Integer Values
1	-63 to 63
2	-8191 to 8191
3	-1048575 to 1048575
4	-134217727 to 134217727
5	Any integer value

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"
```

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
```

USING TTL

`USING TTL` is an optional statement that defines a default time-to-live value for a table's rows. See Using Time to Live in the *Java Direct Driver Developer's Guide* for information on TTL.

If specified, this statement must provide a *ttl* value, which is an integer greater than or equal to 0, followed by a space, followed by time unit designation which is either *hours* or *days*. For example:

```
USING TTL 5 days
```

If `0` is specified, then either `days` or `hours` can be used. A value of `0` causes table rows to have no expiration time. Note that `0` is the default if a default TTL has never been applied to a table schema. However, if you previously applied a default TTL to a table schema, and then want to turn it off, use `0 days` or `0 hours`.

```
USING TTL 0 days
```

Be aware that if you altering an existing table, you can not both add/drop a field *and* alter the default TTL value for the field using the same `ALTER TABLE` statement. These two operations must be performed using separate statements.

Table Creation Examples

The following are provided to illustrate the concepts described above.

```
CREATE TABLE users
COMMENT "This comment applies to the table itself" (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  PRIMARY KEY (id),
)


CREATE TABLE temporary
COMMENT "These rows expire after 3 days" (
  sku STRING,
  id STRING,
  price FLOAT,
```

```
    count INTEGER,
    PRIMARY KEY (sku),
) USING TTL 3 days


CREATE TABLE Users
COMMENT "This is an MR table"(
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  primary key (id)
) IN REGIONS us_east, us_west;


CREATE TABLE usersNoId (
  firstName STRING,
  lastName STRING COMMENT "This comment applies to this field only",
  age INTEGER,
  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,
  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
)


CREATE TABLE myJSON (
    recordID INTEGER,
    jsonData JSON,
    PRIMARY KEY (recordID)
)
```

Modify Table Definitions

Use `ALTER TABLE` statements to either add new fields to a table definition, or delete a currently existing field definition. You can also use an `ALTER TABLE` statement to change the default Time-to-Live (TTL) value for a table, and to add an IDENTITY column to a table, as described in Defining Tables With an IDENTITY Column.

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 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 Option

To delete a field from an existing table, use the `DROP` option:

```
ALTER TABLE table-name (DROP field-name)
```

For example, to drop the `age` field from the `Users` table:

```
ALTER TABLE Users (DROP age)
```

> ✎ **Note:**
>
> You cannot drop a field if it is the primary key, or if it participates in an index.

You can also us the `ALTER TABLE MODIFY FIELD` clause to add, drop, or modify an IDENTITY column in a table. For more information, see Defining Tables With an IDENTITY Column

ALTER TABLE USING TTL

To change the default Time-to-Live (TTL) value for an existing table, use the `USING TTL` statement:

```
ALTER TABLE table-name USING TTL ttl
```

For example:

```
ALTER TABLE Users USING TTL 4 days
```

For more information on the `USING TTL` statement, see USING TTL.

DROP TABLE

To delete a table definition, use the `DROP TABLE` statement:

```
DROP TABLE [IF EXISTS] table-name
```

where:

- `IF EXISTS` is optional. If you use this option and the specified table does not exist, the system returns a message:

  ```
  sql-> drop table if exists foo;
  Statement did not require execution
  ```

  If you do not specify `IF EXISTS` , and the table does not currently exist, the DROP statement returns as follows:

  ```
  sql-> drop table foo;
  Error handling command drop table foo:
  Error: User error in query: DROP TABLE failed for table foo:
  Table does not exist: foo
  ```

  If you specify `IF EXISTS`, and the table exists, the DROP statement executes successfully:

  ```
  sql-> create table foo (first string, second integer, primary key
  (second));
  Statement completed successfully
  sql-> drop table if exists foo;
  Statement completed successfully
  ```

- *table-name* is the name of the table you want to drop.

As soon as you execute the DROP TABLE statement, users can no longer access the deleted table or its data. Deleting all of the table data occurs asynchronously in the background after you execute the DROP TABLE statement.

If you choose to drop an MR Table in a particular region, it still continues to remain an MR Table in the other participating regions. In a case where you want to drop a particular MR Table from multiple regions, you must execute the `DROP TABLE` statement in each region separately.

> **Note:**
>
> In a case where an MR Table is dropped in all remote regions but still exists in local region, it still continues to be an MR Table linked with a single region. Such an MR Table with a single region can be expanded to more regions in future. That is, you can add new regions to this table in future, as needed.

If the table to drop has child tables, you must drop those first. For example, if you have these tables:

- myTable
- myTable.childTable1
- myTable.childTable2

You must first drop `myTable.childTable1` and `myTable.childTable2` before you can drop the parent `myTable`. If you try to drop the parent, the statement returns an error.

CREATE INDEX

To add an index definition to the store, use a `CREATE INDEX` statement. It can be used to create simple indexes and multi-key indexes. It can also be used to create JSON indexes.

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
- Number
- Float
- Double
- Json

  Note that there are some differences and restrictions on indexing Json data versus other data types. See JSON Indexes for more information.

- • 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.

Simple Indexes

An index is simple if it does not index any maps or arrays. To create a simple index:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (path_list)
```

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.

- • *path_list* is a comma-separated list of one or more *name_paths*. A name_path refers to an element of a table. Normally these are schema fields — that is, field names that appear in the CREATE TABLE expression used to create the associated table.

  However, if the table contains a record, then the name_path may be record keys that use dot-notation to identify a record field. For example:

```
CREATE TABLE example (
        id INTEGER,
        myRecord RECORD(field_one STRING, field_two STRING),
        PRIMARY KEY (id)
    )
```

  An index can then be created on field_one by using the name_path of `myRecord.field_one`. See Indexing Embedded Records for a more detailed explanation of indexing records.

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.

Multi-Key Indexes

Multi-key indexes are used to index all the elements of an array. They are also used to index all of the elements and/or values of a map.

For each table row, a multi-key index contains as many entries as the number of elements/entries in the array/map that is being indexed (although duplicate entries are not represented in the index). To avoid an explosion in the number of index entries, only one array/map may be contained in a single multi-key index.

To create a multi-key index, use one of the following forms:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path.keys())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path.values())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(name-path.keys(),name-path.values())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path[])
```

The syntax shown, above, is identical to that described in Simple Indexes, with the following additions:

- `.keys()`

  The index is created on the keys in a map. If used, *name-path* must be a map.

- `.values()`

  The index is created on the values in a map. If used, *name-path* must be a map.

- `[]`

  The index is created on an array. If used, *name-path* must be array.

For each of the previously identified forms, a comma-seperated list of name-paths may be provided. Some restrictions apply.

Multi-Key Index Restrictions

The following restrictions apply to multi-key indexes:

- There is at least one name-path that uses a multi-key step (`.keys()`, `.values()`, or `[]`). Any such path is called a *multi-key path*, and the associated index field a *multi-key field*. The index definition may contain more than one multi-key path, but all multi-key paths must use the same name-path before their multi-key step.

- Any non-multi-key paths must be simple paths.

- The combined path specified for the index must contain at least one map and/or array. These must contain indexable atomic items, or record items, or map items. That is, an index of an array of arrays is not supported, nor is an index of maps containing arrays.

  For example, given the following table definition:

  ```
  create table Foo (
  id INTEGER,
  complex1 RECORD(mapField MAP(ARRAY(MAP(INTEGER)))),
  complex2 RECORD(matrix ARRAY(ARRAY(RECORD(a LONG, b LONG)))
  primary key(id)
  )
  ```

  The path expression `complex2.matrix[]` is not valid, because the result of this path expression is a sequence of arrays, not atomic items. Neither is `complex2.matrix[][].a` valid, because you cannot index arrays inside other arrays (in fact this path will raise a syntax error, because the syntax allows at most one `[]` per index path).

  On the other hand, the path `complex1.mapField.someKey[].someOtherKey` is valid. In this case, the path `complex1.mapField.someKey` specifies an array containing maps, which is valid. Notice that in this index path, `someKey` and `someOtherKey` are map-entry keys. So, although we are indexing arrays that are contained inside maps, and the arrays being indexed contain maps, the path is valid, because it is selecting specific entries from the map, rather than indexing all the map entries in addition to all the array entries.

- If the index is indexing an array-valued field:
  - If the array contains indexable atomic items:
    * There must be a single multi-key index path of the form `M[]` (without any name_path following after the []). Again, this implies that you cannot index more than one array in the same index.
    * For each table row (R), a number of index entries are created as follows:

      The simple index paths (if any) are computed on R.

      Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M.

      Finally, for each value (V) returned by M[], an index entry is created whose field values are V and the values of the simple paths.
    * Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.

- If the array contains records or maps:

  * All of the multi-key paths must be of the form `M[].name_path`. Each name_path appearing after M[] in the multi-key index path must return at most one indexable atomic item.

  * For each table row (R), a number of index entries are created as follows:

    The simple index paths (if any) are computed on R.

    Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M.

    Next, for each value (V) returned by M[], one index entry is created as follows:

    The elements contained in each V are computed. This returns a single indexable atomic item (which may be the NULL or EMPTY item). An index entry is created for each of these, whose field values are the values of the simple index paths plus the values found for element contained in V.

  * Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.

- If the index is indexing a map-valued field, the index may be indexing only map keys, or only map elements, or both keys and elements. In all cases, the definition of map indexes can be given in terms of array indexes, by viewing maps as arrays containing records with 2 fields: a field with name "key" and value a map key, and a field named "element" and value the corresponding map element (that is, MAP(T) is viewed as ARRAY(RECORD(key STRING, element T))). Then, the 2 valid kinds for map indexes are:

  1. A single multi-key index path using a keys() step. Using the array view of maps, M.keys() is equivalent to M[].key.

  2. One or more multi-key index paths, all using a .values() step. If Ri is an value contained in the map, then each of these has the form M.values().Ri. Using the array view of maps, each M.values().Ri path is equivalent to M[].element.Ri.

JSON Indexes

An index is a JSON index if it indexes at least one field that is contained inside JSON data.

Because JSON is schema-less, it is possible for JSON data to differ in type across table rows. However, when indexing JSON data, the data type must be consistent across table rows or the index creation will fail. Further, once or more JSON indexes have been created, any attempt to write data of an incorrect type will fail.

Indexing JSON data and working with JSON indexes is performed in much the same way as indexing non-JSON data. To create the index, specify a path to the JSON field using dot notation.

When creating JSON indexes, you must specify the data's type, using the `AS` keyword. The data type must be atomic, and cannot be a float. That is, only integer, long,

double, number, string, and boolean are supported types for JSON indexes. Note that arrays and maps can be indexed so long as they contain these atomic values.

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(JSONRow.JSONField AS data_type)
```

When creating a multi-key index on a JSON map, a type must not be given for the .keys() expression because the type will always be String. However, a type declaration is required for the .values() expression. Beyond that, all the constraints described in Multi-Key Index Restrictions also apply to a JSON multi-keyed index.

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(JSONRow.JSONField.keys(),\
    JSONRow.JSONField.values() AS data_type)
```

CREATE FULL TEXT INDEX

To create a text index on that table that indexes the category and txt columns, use CREATE FULLTEXT INDEX statement:

```
CREATE FULLTEXT INDEX [if not exists] <index-name> ON <table-name>
(<field-name> [ <mapping-spec> ], ...)
[ES_SHARDS = <n>] [ES_REPLICAS = <n>]
```

For example:

```
kv-> execute 'CREATE FULLTEXT INDEX JokeIndex
ON Joke (category, txt)'
Statement completed successfully
```

While creating index, CREATE FULLTEXT INDEX statement uses the OVERRIDE flag, which allows to delete any index existing in Elasticsearch by the same name as would be created by the command.

```
CREATE FULLTEXT INDEX [IF NOT EXISTS] index_name ON table_name
    (field_name [{mapping_spec}] [, field_name [{mapping_spec}]]...)
    [ES_SHARDS = value] [ES_REPLICAS = value]
    [OVERRIDE] [COMMENT comment]
```

For example:

```
CREATE INDEX JokeIndex on  Joke (category, txt) OVERRIDE
```

For more information, see Creating Full Text Index and Mapping a Full Text Index Field to an Elasticsearch Field in the *Integrations Guide*.

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
```

Use `DROP INDEX` on a text index to stop the population of the index from NoSQL Database shards, and removes the mapping and all related documents from Elasticsearch. See the following statement:

```
DROP INDEX [IF EXISTS] index_name ON table_name
```

For example:

```
kv-> execute 'DROP INDEX JokeIndex on Joke'
    Statement completed successfully
```

While deleting index, you can use the `OVERRIDE` flag. The `DROP INDEX` statement uses the `OVERRIDE` flag to enable overriding of the default constraints:

```
DROP INDEX [IF EXISTS] index_name ON table_name [OVERRIDE]
```

For example:

```
DROP INDEX JokeIndex on Joke OVERRIDE
```

For more information, see Deleting FTI in the *Integrations Guide*.

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.

  For Map and Array fields, use `[]` 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.

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`

  Be aware that the host names specified here must be resolvable using DNS or the local `/etc/hosts` file.

- `-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 for details.

Securing Oracle NoSQL Database Proxy Server

If configured properly, the proxy can access a secure installation of Oracle NoSQL Database. Be aware that the Proxy Server itself does not support secure connections from Oracle NoSQL Database clients. Those connections are always performed in the clear and without authentication. However, the Proxy Server can connect to a secure store, and that connection will be both authenticated and encrypted.

To do this, the `-username` and `-security` proxy options must be specified when you start the proxy server. If you are using Oracle NoSQL Database Enterprise Edition, then you might be using Oracle wallet (this is what kvlite uses when it is obtained from an Enterprise Edition package). If so, then you must also use the `oraclepki.jar` file when you start the proxy server.

When starting the proxy server, the `-security` option identifies the security file you want to use. If you are using kvlite, then it will create a security file for you when it creates a fresh `KVROOT` directory. In this case, use `KVROOT/security/user.security`. If you are not using kvlite, but instead are performing a full secured installation of Oracle NoSQL Database, then you create a copy of `KVROOT/security/client.security`, and add additional parameters that are necessary for a client to connect securely to the store. See SSL model in the *Security Guide*.

Either way, you should have a file on the same host as your proxy server will run that looks something like this (notice that this installation uses Oracle wallet):

```
oracle.kv.ssl.trustStore=client.trust
oracle.kv.transport=ssl
oracle.kv.auth.username=admin
oracle.kv.ssl.protocols=TLSv1.2,TLSv1.1,TLSv1
oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
oracle.kv.auth.wallet.dir=user.wallet
```

To run KVProxy and access the secure Oracle NoSQL Database deployment, do the following (notice that because we are using Oracle wallet in this example, `oraclepki.jar` is specified):

```
java -cp <KVHOME>/lib/kvclient.jar: \
<KVPROXY>/lib/kvproxy.jar:<KVHOME>/lib/oraclepki.jar \
oracle.kv.proxy.KVProxy -helper-hosts node01:5000 -port 5010 \
-store mystore -username admin -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 configure the firewall for the proxy server to limit access to 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.

C

Third Party Licenses

The Oracle NoSQL Database Client is licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:.

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The following applies to this Oracle NoSQL client as well as all other products under the Apache 2.0 license.

Apache License Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original

work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b. You must cause any modified files to carry prominent notices stating that You changed the files; and

   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the

Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions or use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Licensing terms for SLF4J

SLF4J source code and binaries are distributed under the MIT license.

Copyright © 2004, 2013 QOS.ch. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

*   The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

*   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.