Oracle NoSQL Database

Python Driver Quick Start

12c Release 1

(Library Version 12.1.3.3)

Legal Notice

Table of Contents

Introduction 3
Installation 3
Using the Proxy Server 3
The nosqldb Python Module 4
Connecting to the Store 5
    Automatically Starting the Proxy Server 6
Creating Table and Index Definitions 7
Writing to a Table Row 8
Deleting a Table Row 9
Reading a Single Table Row 9
Reading Multiple Table Rows 10
Reading Using Indexes 12
Sequence Execution 14
Setting Consistency Guarantees 16
Setting Durability Guarantees 18
Proxy Server Reference 19
    Securing Oracle NoSQL Database Proxy Server 20
    Trouble Shooting the Proxy Server 24

Introduction

This article provides a quick introduction to the Oracle NoSQL Database Python driver. This driver provides Python client access to data stored in Oracle NoSQL Database tables.

To work, the Python driver requires use of a proxy server which translates network activity between the Python client 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 Python 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).

This quick start assumes that you have read and understood the concepts described in the *Oracle NoSQL Database Getting Started with the Table API* guide. You can find that guide in your Oracle NoSQL Database server installation package, or find it here:

- HTML:

  http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuideTables/index.html

- PDF:

  http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuideTables/Oracle-NoSQLDB-GSG-Tables.pdf

The entirety of the API used by the Python driver is described in the *Python API Reference Guide*.

Installation

The nosqldb driver supports Python 2.6 and 2.7.

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

```
pip install nosqldb
```

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

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

```
pip install thrift
```

Using the Proxy Server

The proxy server is a Java application that accepts network traffic from the Python API, 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. The proxy server and all associated jar files are bundled with the nosqldb package, so the location of these are relative to your python installation directory:

```
<python-site-packages-directory>/nosqldb/kvproxy/lib
```

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`

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

- `-port`

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

- `-store`

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

A range of other command line options are available. In particular, if you are using the proxy server with a secure store, you must provide authentication information to the proxy server. In addition, you will probably have to identify a store name to the proxy server. For a complete description of the proxy server and its command line options, see Proxy Server Reference (page 19).

The simple examples provided in this quick start guide were written to work with an proxy server that is connected to a `kvlite` instance. The location of the `kvclient.jar` and `kvproxy.jar` files were provided using a `CLASSPATH` environment variable. The command line call used to start the proxy server was:

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

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

1. Identify the store's name, host and port using a `StoreConfig` instance. The host and port that you provide is for any machine hosting a node in the store. Because the store is comprised of many hosts, there should be multiple host/port pairs for you to choose from. You provide this information to `StoreConfig` using an array of strings in the form `'hostname:port'`.

2. Identify the host and port where the proxy is running. This is done by providing a `'hostname:port'` string to the `Factory.open()` method.

   ### Note

   At least one of the host:port pairs provided for the store must match a host:port pair provided to the proxy server, otherwise the connection to the proxy server will fail.

For example, suppose you have a Oracle NoSQL Database store named "MyNoSQLStore" 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('MyNoSQLStore', [storehost])
        return Factory.open(proxy, kvstoreconfig)
    except ConnectionException, ce:
        logging.error("Store connection failed.")
        logging.error(ce.message)
        sys.exit(-1)
```

If you are using a secure store, then your proxy server must first be configured to authenticate to the store. See Securing Oracle NoSQL Database Proxy Server (page 20) for details.

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

```
# configure and open the store
def open_store():
    try:
        kvstoreconfig = StoreConfig('MyNoSQLStore', [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)
```

`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()
```

Automatically Starting the Proxy Server

Your client code can start the proxy server on the local host when it opens a connection to the store. To do this, your driver must be able to locate the `kvclient.jar` and `kvproxy.jar` files on your local host. These are automatically installed when you install the nosqldb driver, so you should not normally need to do anything extra in order to have the driver automatically start the proxy server.

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

1. If they are specified as parameters to the `ProxyConfig` constructor, then that location is used.
2. If that information is not specified to the constructor, then it is taken from the `KVSTORE_JAR` and `KVPROXY_JAR` environment variables.

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

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

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

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

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

    kvstoreconfig = StoreConfig('kvstore', [kvlite])
    kvproxyconfig = ProxyConfig()

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

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

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

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

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

For information on configuring your proxy server to connect to a secure store, see Securing Oracle NoSQL Database Proxy Server (page 20).

Creating Table and Index Definitions

Before you can write data to tables in your store, you must define your tables using table DDL statements. You also use DDL statements to define indexes. The table DDL is described in detail in the *Oracle NoSQL Database Getting Started with the Table API* guide.

If you want to submit table DDL statements to the store from your Python client code, use either `Store.execute_sync()` or `Store.execute()`. The latter function submits DDL statements to the store asynchronously, which you may want to do when creating indexes or dropping tables because these operations can take a long time.

For example, to drop a table and the recreate it asynchronously:

```
def do_store_ops(store):
    ### Drop the table if it exists
    try:
        ddl = """DROP TABLE IF EXISTS Users2"""
        store.execute_sync(ddl)
```

```
        logging.debug("Table drop succeeded")
    except IllegalArgumentException, iae:
        logging.error("DDL failed.")
        logging.error(iae.message)
        return

    ### Create a table
    try:
        ddl = """CREATE TABLE Users2 (
            id INTEGER CHECK(id > 300),
            firstName STRING,
            lastName STRING,
            runner ENUM(slow,fast) DEFAULT slow,
            myRec RECORD(a STRING),
            myMap MAP(INTEGER CHECK(ELEMENTOF(myMap) > 500)),
            myArray ARRAY(INTEGER),
            myBool BOOLEAN DEFAULT FALSE,
            PRIMARY KEY (SHARD(id, firstName), lastName)
        )"""
        store.execute_sync(ddl)
        logging.debug("Table creation succeeded")
    except IllegalArgumentException, iae:
        logging.error("DDL failed.")
        logging.error(iae.message)
```

Writing to a Table Row

Once you have defined a table in the store, use `Store.put()` to write table rows to the store. To do this, you define a dictionary that includes all the row data, and then give that to a `Row` instance, which you then use with `Store.put()`. For example, suppose you have a table designed like this:

```
CREATE TABLE myTable (
    item STRING,
    description STRING,
    count INTEGER,
    percentage FLOAT,
    PRIMARY KEY (item)
)
```

You can write a row of table data in the following way:

```
def do_store_ops(store):
    row_d = {  'item' : 'bolts',
               'description' : "Hex head, stainless",
               'count' : 5,
               'percentage' : 0.2173913}
    row = Row(row_d)
    try:
        store.put("myTable", row)
        logging.debug("Store write succeeded.")
```

```
    except IllegalArgumentException, iae:
        logging.error("Could not write table.")
        logging.error(iae.message)
```

Deleting a Table Row

Use `Store.delete()` to delete a table row. To do this, you use a dictionary to define the full primary key for the row that you want to delete. For example:

```
def do_store_ops(store):
    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)
```

If the deletion operation was successful, this method returns the tuple:

```
(True, <OldRow>)
```

where `<OldRow>` is the row that was deleted. If the deletion operation was not successful, this method returns:

```
(False, None)
```

Reading a Single Table Row

To read a single table row, create a dictionary that defines the full primary key, then use it with the `Store.get()` method. This method returns a dictionary with keys that are identical to the row's column names.

For example, to retrieve the table row created in Writing to a Table Row (page 8):

```
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)
```

Reading Multiple Table Rows

Use `Store.multi_get()` or `Store.table_iterator()` to read multiple rows from a table at a time. `Store.multi_get()` requires you to provide a shard key in the form of a dictionary. `Store.table_iterator()` requires a partial primary key (which could also be a shard key). The example provided here uses `Store.multi_get()`. If all of the shard keys are not present, then the function will throw an `IllegalArgumentException`.

For example, suppose you design 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)
)
```

And you populate it with data like this:

```
def do_store_ops(store, item_type, item_cat, item_class,
                 item_color, item_size, price, inv_count):
    row_d  = { 'itemType' : item_type,
               'itemCategory' : item_cat,
               'itemClass' : item_class,
               'itemColor' : item_color,
               'itemSize'  : item_size,
               'price'     : price,
               'inventoryCount' : inv_count
               }
```

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


if __name__ == '__main__':

    ...

    do_store_ops(store, "Hats", "baseball", "longbill",
                 "red", "small", 12.07, 127)
    do_store_ops(store, "Hats", "baseball", "longbill",
                 "red", "medium", 13.07, 201)
    do_store_ops(store, "Hats", "baseball", "longbill",
                 "red", "large", 14.07, 309)

    ...
```

Then you can retrieve all of the rows in the table by providing just the shard key because, in this example, the shard key is identical for all the rows in the table. (Normally, if you wanted to display all the rows in a table, you would use `Store.table_iterator()` with an empty row for the `key` parameter.)

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

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

        row_list =
            store.multi_get("myTable",   # table name
                            False,       # Retrieve only keys?
```

```
                                shard_key_d) # partial primary key
        if not row_list:
            logging.debug("Table retrieval failed")
        else:
            logging.debug("Table retrieval succeeded.")
            for r in row_list:
                display_row(r)
    except IllegalArgumentException, iae:
        logging.error("Table retrieval failed.")
        logging.error(iae.message)
```

Reading Using Indexes

Use `Store.index_iterator()` to read table rows based on a specified index. To use this function, the index must first be created using the `CREATE INDEX` statement.

There are two ways to identify the index values you want the results set based on. The first way is to provide a dictionary representing the indexed field(s) and value(s) that you want retrieved. The second way is to provide a `FieldRange` which identifies the starting and ending index values you want returned. Both of these mechanisms can be used together to restrict the return set values.

If both the dictionary and field ranges are `None`, then every row in the table matching the specified index is contained in the return set.

For example, suppose you have a table defined 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))
```

With this index:

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

And you populate the table with data like this:

```
def do_store_ops(store, surname, famName, uid, phone, address,
                 email, DoB):
    row_d =  { 'surname' : surname,
               'familiarName' : famName,
               'userID' : uid,
               'phonenumber' : phone,
               'address'  : address,
               'email'     : email,
               'dateOfBirth' : DoB
               }
```

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

if __name__ == '__main__':

    ...

    do_store_ops(store, "Anderson", "Pete", "panderson",
                 "555-555-5555", "1122 Somewhere Court",
                 "panderson@example.com", "1994-05-01")
    do_store_ops(store, "Andrews", "Veronica", "vandrews",
                 "666-666-6666", "5522 Nowhere Court",
                 "vandrews@example.com", "1973-08-21")
    do_store_ops(store, "Bates", "Pat", "pbates",
                 "777-777-7777", "12 Overhere Lane",
                 "pbates@example.com", "1988-02-20")
    do_store_ops(store, "Macar", "Tarik", "tmacar",
                 "888-888-8888", "100 Overthere Street",
                 "tmacar@example.com", "1990-05-17")

    ...
```

Then you can read using the `DoB` index using the following function. In the following example, `BLOCK 1` (see the comments in the code) is commented out, because its usage with `BLOCK 2` causes an `IllegalArgumentException` to be raised.

```
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 = {}
```

```
## BLOCK 1:
#  Uncomment this block to look up only table rows with a
# dateOfBirth field set to "1988-02-20". If this block and
# BLOCK 2 are both used, then and IllegalArgumentException is
# raised because a FieldRange cannot be used with a full index
# key. (It can be used with a partial index key.)
#
# key_d = {"dateOfBirth" : "1988-02-20"}
# mro = None

## BLOCK 2:
# This field range restricts the results set to only
# those rows with a dateOfBirth field value between
# "1990-01-01" and "2000-01-01", inclusive.
#
# Do not use this with BLOCK 1.
field_range = FieldRange({
                          ONDB_FIELD : "dateOfBirth",
                          ONDB_START_VALUE : "1990-01-01",
                          ONDB_END_VALUE : "2000-01-01",
                          # 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)
```

Sequence Execution

You can create a sequence of write operations that are atomically executed. To do this, create an array of `Operation` objects, each element of which represents a single write operation. Each `Operation` object is created using a dictionary that represents the operation type (`PUT` or `DELETE`, for example), the name of the table to operate on, and other necessary

information such as a new table row to write, and whether to abort if the operation is unsuccessful.

The sequence is executed using `Store.execute_updates()`.

For example, if you had a table populated with data such as is described in Reading Multiple Table Rows (page 10), then you could update the price and inventory values for each row of the table in an atomic operation like this:

```
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 : if_unsuccess
                   })
    op_array.append(op)

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)

    ...
```

Setting Consistency Guarantees

By default, read operations are performed with a consistency of guarantee of `NONE_REQUIRED`. You can change this by defining a new consistency guarantee using a `Consistency` class object, and then passing that to `Store.get()` using a `ReadOptions` class object.

For convenience, the nosqldb module pre-defines some simple consistency guarantees so that you do not have to configure the guarantee yourself:

```
NONE_REQUIRED =
    Consistency({ONDB_SIMPLE_CONSISTENCY: 'NONE_REQUIRED'})
ABSOLUTE = Consistency({ONDB_SIMPLE_CONSISTENCY: 'ABSOLUTE'})
NONE_REQUIRED_NO_MASTER = Consistency({
```

```
    ONDB_SIMPLE_CONSISTENCY: 'NONE_REQUIRED_NO_MASTER'})
```

For example, the code fragment shown in Reading a Single Table Row (page 9) can be rewritten to use a simple consistency policy in the following way:

```
from nosqldb import Consistency
from nosqldb import ConsistencyException
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import ProxyConfig
from nosqldb import ReadOptions
from nosqldb import RequestTimeoutException
from nosqldb import StoreConfig

#### constants required for ReadOptions
from nosqldb import ONDB_CONSISTENCY
from nosqldb import ONDB_TIMEOUT
## possible consistency guarantees:
from nosqldb import NONE_REQUIRED
from nosqldb import ABSOLUTE
from nosqldb import NONE_REQUIRED_NO_MASTER

...

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):
    ## 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)
```

Setting Durability Guarantees

By default, write operations are performed with a durability guarantee of `COMMIT_NO_SYNC`. You can override this by creating and using a durability guarantee.

Use a `Durability` class object to define your preferred durability guarantee. You then pass this object to a `WriteOptions` class object which is then provided to the `Store.put()` method.

For example, the code fragment shown in Writing to a Table Row (page 8) can be rewritten to use a durability policy in the following way:

```
from nosqldb import Durability
from nosqldb import DurabilityException
from nosqldb import Factory
from nosqldb import IllegalArgumentException
from nosqldb import RequestTimeoutException
from nosqldb import Row
from nosqldb import WriteOptions

...

##### Constants needed for the write options
from nosqldb import ONDB_DURABILITY
from nosqldb import ONDB_MASTER_SYNC
from nosqldb import ONDB_REPLICA_SYNC
from nosqldb import ONDB_REPLICA_ACK
from nosqldb import ONDB_TIMEOUT
##      For Durability, could use one of
##      COMMIT_SYNC, COMMIT_NO_SYNC, or
##      COMMIT_WRITE_NO_SYNC

...

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

    ## This Durability is the same as you get if using
    ## COMMIT_SYNC. We do it the long way here for illustrative
    ## and readability purposes.
    dur = Durability({ONDB_MASTER_SYNC : 'SYNC',
                      ONDB_REPLICA_SYNC : 'NO_SYNC',
                      ONDB_REPLICA_ACK : 'SIMPLE_MAJORITY'})

    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)
```

Proxy Server Reference

The proxy server command line options are:

```
nohup java -cp KVHOME/lib/kvclient.jar:kvproxy/lib/kvproxy.jar
oracle.kv.proxy.KVProxy -help
 -port <port-number> Port number of the proxy server. Default: 5010
 -store <store-name> Required KVStore name. No default.
 -helper-hosts <host:port,host:port,...>   Required list of KVStore
       hosts and ports (comma separated).
 -security <security-file-path>  Identifies the security file used
       to specify properties for login. Required for connecting to
       a secure store.
  -username <user>  Identifies the name of the user to login to the
       secured store. Required for connecting to a secure store.
  -read-zones <zone,zone,...>  List of read zone names.
  -max-active-requests <int> Maximum number of active requests towards
       the store.
  -node-limit-percent <int> Limit on the number of requests, as a
       percentage of the requested maximum active requests.
  -request-threshold-percent <int> Threshold for activating request
       limiting, as a percentage of the requested maximum active
       requests.
  -request-timeout <long> Configures the default request timeout in
       milliseconds.
  -socket-open-timeout <long> Configures the open timeout in
```

```
      milliseconds used when establishing sockets to the store.
-socket-read-timeout <long> Configures the read timeout in
      milliseconds associated with the underlying sockets to the
      store.
-max-iterator-results <long> A long representing the maximum
      number of results returned in one single iterator call.
      Default: 100
-iterator-expiration <long>  Iterator expiration interval in
      milliseconds.
-max-open-iterators <int>    Maximum concurrent opened iterators.
      Default: 10000
-num-pool-threads <int>      Number of proxy threads. Default: 20
-max-concurrent-requests <int>      The maximum number of
      concurrent requests per iterator. Default: <num_cpus * 2>
-max-results-batches <int>      The maximum number of results
      batches that can be held in the proxy per iterator.
      Default: 0
-help  Usage instructions.
-version  Print KVProxy server version number.
-verbose  Turn verbose flag on.
```

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

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

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

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

- `-security`
- `-username`

Note

Drivers are able to start and stop the proxy server on the local host if properly configured. See Automatically Starting the Proxy Server (page 6) for details.

Securing Oracle NoSQL Database Proxy Server

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

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

To install Oracle NoSQL Database securely:

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

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

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

3. Enter a password for your store and then reenter it for verification. In this case, the password file is used, and the `securityconfig` tool will automatically generate the following security related files:
   ```
   Enter a password for the Java KeyStore: ***********
   Re-enter the KeyStore password for verification: ***********
   Created files:
   security/client.trust
   security/client.security
   security/store.keys
   security/store.trust
   security/store.passwd
   security/security.xml
   ```

   ## Note

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

4. Start the Storage Node Agent (SNA):
   ```
   nohup java -Xmx256m -Xms256m \
   -jar KVHOME/lib/kvstore.jar start -root KVROOT&
   ```

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

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

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

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

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

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

```
kv-> plan deploy-zone -name mydc -rf 1 -wait
Executed plan 2, waiting for completion...
Plan 2 ended successfully
kv-> plan deploy-sn -zn zn1 -port 5000 -host node01 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
kv-> plan deploy-admin -sn sn1 -port 5001 -wait
Executed plan 4, waiting for completion...
Plan 4 ended successfully
kv-> pool create -name mypool
kv-> pool join -name mypool -sn sn1
Added Storage Node(s) [sn1] to pool mypool
kv-> topology create -name mytopo -pool mypool -partitions 30
Created: mytopo
kv-> plan deploy-topology -name mytopo -wait
Executed plan 5, waiting for completion...
Plan 5 ended successfully
```

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

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

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

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

```
Re-enter the secret value for verification: ********
Secret created
OK
```

Note

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

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

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

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

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

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

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

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

Note

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

Trouble Shooting the Proxy Server

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Start the proxy with the following parameter:

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

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

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

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