Oracle® NoSQL Database

C Table Driver Developer's Guide

Release 20.2

E85383-09

August 2020

Introduction

This article provides a quick introduction to the Oracle NoSQL Database C driver. This driver provides native C client access to data stored in Oracle NoSQL Database tables. (There is a C JNI driver which provides a Key/Value API for access to Oracle NoSQL Database data. That driver relies on a JNI layer and is not described by this article.)

The C driver is available as a separate download from the Oracle NoSQL Database server package. You can obtain both the server and the driver download packages from Oracle Technology Network.

To work, the C driver requires use of a proxy server which translates network activity between the C 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 C 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 *SQL Reference Guide*.

The entirety of the API used by the C driver is described in the *C Table Driver API Reference*.

Installation

Both the C driver and the proxy are available in a common download package. The proxy server resides in the `kvproxy` directory, and is provided as a Java jar file (`kvproxy.jar`). To use the proxy, you must also have a Oracle NoSQL Database server installation. Specifically, the `kvclient.jar` file from that installation must be available to the proxy.

The C driver library must be compiled before it can be used by your client code. The source code for this library and library dependencies are available in the `c` directory of the driver package. The compile instructions are available in the Building Oracle NoSQL Database C Driver for Tables tutorial.

The C driver is also available as a pre-compiled binary. To access this binary, download and install the relevant rpm or deb file. If you use the rpm or deb, the required Java jar files are also installed on your system. In this case, the relevant library and jar files are installed into `/usr/local/lib`. To use them, you might need to adjust your `CLASSPATH` and `LD_LIBRARY_PATH` environment variables to include that directory.

Using the Proxy Server

The proxy server is a Java application that accepts network traffic from the Table C 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 C 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:

- *KVHOME*/lib/kvclient.jar
- .../kv-c-driver-*X.Y.Z*/kvproxy/lib/kvproxy.jar

> **Note:**
>
> If you installed using rpm or deb, then these files are located in `/usr/local/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.

The simple examples provided in this quick start guide were written to work with an proxy server that is connected to a `kvlite` instance which was started with default values. 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 -port 7010 \
-helper-hosts localhost:5000 -store kvstore
```

Compiling and Running C Clients

To compile your C clients, link either `libkvstore.so` or `libkvstore-static.a`. They can be found in the `lib` directory contained in the installation location that you provided to `cmake`.

If you use the `.so` file, make sure to add its installation directory to your `LD_LIBRARY_PATH` environment variable so that the library can be found at run time.

The `kvstore.h` header file is located in the `include` directory contained in the installation location that you provided to `cmake`.

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 `kv_config_t` structure. The host and port that you provide to this structure 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 create the `kv_config_t` structure using `kv_create_config()`. You can release the structure using `kv_release_config()`.

2. Identify the host and port where the proxy is running. You do this using `kv_open_store()`. This function creates an `kv_store_t` structure, which is what you will use for all subsequent store operations. You release this structure using `kv_close_store()`.

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:

```
#include <stdlib.h>
#include <stdio.h>
#include "kvstore.h"

void open_store(kv_store_t **);
void do_store_ops(kv_store_t *);
```

```
int main(void) {
    kv_store_t *store = NULL;

    open_store(&store);
    if (!store) {
        goto ERROR;
    }

    do_store_ops(store);


ERROR:
    /* Close the store handle. */
    if (store) {
        kv_close_store(&store);
    }
    return 0;
}

void do_store_ops(kv_store_t *store)
{
    printf("Do store operations here.\n");
}

void
open_store(kv_store_t **store)
{
    kv_config_t *config = NULL;
    kv_error_t ret;

    ret = kv_create_config("kvstore",   // store name
                           "localhost", // host name
                           5000,        // host port
                           &config);

    if (ret != KV_SUCCESS) {
        return;
    }

    /* Connect to a proxy server */
    ret = kv_open_store(store, "localhost", 7010, config);
    if (ret != KV_SUCCESS) {
        printf("could not connect to the store.\n");
        // Release the configuration structure
        kv_release_config(&config);
    }
}
```

If you are using a secure store then the configuration of your store handle must also include the user name. Use `kv_config_set_auth_user()` for this purpose.

Automatically Starting the Proxy Server

Your client code can start the proxy server on the local host when it opens the store using `kv_open_store_with_proxy()`. This function requires everything that `kv_open_store()` requires, plus a `kv_proxy_config_t` structure populated with the on-disk location of the `kvclient.jar` and `kvproxy.jar` files.

You populate the `kv_proxy_config_t` structure using `kv_create_proxy_config()`. You can release this structure (only necessary if some error occurs when opening the store) using `kv_release_proxy_config()`. When your code is done with the proxy server, shut it down using `kv_shutdown_proxy()`.

For example:

```
int main(void) {
    kv_store_t *store = NULL;

    open_store(&store);
    if (!store) {
        goto ERROR;
    }

    do_store_ops(store);


ERROR:
    /* Close the store handle. */
    if (store) {
        kv_shutdown_proxy(store);
        kv_close_store(&store);
    }
    return 0;
}

// do_store_ops() not implemented in this example

void
open_store(kv_store_t **store)
{
    kv_config_t *config = NULL;
    kv_proxy_config_t *proxy_config = NULL;
    kv_error_t ret;

    const char *path2kvclient = "/export/kvstore/lib/kvclient.jar";
    const char *path2kvproxy =
                    "/export/c_driver/kvproxy/lib/kvproxy.jar";

    ret = kv_create_config("kvstore",   // store name
                           "localhost", // host name
                           5000,        // host port
                           &config);
            // All configs are correct for kvlite
    if (ret != KV_SUCCESS) {
        return;
    }

    // Create the proxy configuration structure.
    // This must identify where the two relevant jar
    // files reside on disk.
    ret = kv_create_proxy_config(path2kvclient,
                                 path2kvproxy,
                                 &proxy_config);
    if (ret != KV_SUCCESS) {
        printf("could not create proxy config.\n");
        return;
    }

    ret = kv_open_store_with_proxy(store,
                                   "localhost",
                                   7010,
                                   config,
                                   proxy_config);
    if (ret != KV_SUCCESS) {
        printf("could not connect to the store.\n");
        // Release the configuration structure
        if (config) {
            kv_release_config(&config);
        }

        if (proxy_config) {
            kv_release_proxy_config(&proxy_config);
        }
        store = NULL;
    }
}
```

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 *SQL Reference Guide*.

If you want to submit table DDL statements to the store from your C client code, use either `kv_table_execute_sync()` or `kv_table_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 create a table synchronously:

```
void do_store_ops(kv_store_t *store)
{
    kv_error_t ret;
    /* ... Data operations ... */
    kv_statement_result_t *result = NULL;

    const char *statement = "CREATE TABLE Users2 (\
        id INTEGER, \
        firstName STRING, \
        lastName STRING, \
        description STRING, \
        PRIMARY KEY (SHARD(id, firstName), lastName)\
    )";

    ret = kv_table_execute_sync(store, statement, &result);
    if (ret != KV_SUCCESS) {
        printf("Table creation failed.\n");
        printf("Error message is %s\n",
            kv_statement_result_get_error_message(result));
    } else {
        printf("Table create succeeded.\n");
    }
}
```

Writing to a Table Row

Once you have defined a table in the store, use `kv_create_row()` to create an empty table row. Then use the appropriate `kv_row_put_xxx()` function (where `xxx` is the data type for the field that you are writing) to populate each field with data. Finally, use `kv_table_put()` to actually write the table row to the store. For example, for 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 fashion (the store open and close is skipped for brevity):

```
void
do_store_ops(kv_store_t *store)
{
    kv_error_t ret;
    kv_row_t *row = NULL;

    row = kv_create_row();
    if (!row) {
       printf("row creation failed.\n");
       goto cleanup;
    }
    ret = kv_row_put_string(row, "item", "Bolts");
    if (ret != KV_SUCCESS) {
        printf("row put 'item' failed.\n");
        goto cleanup;
    }
    ret = kv_row_put_string(row, "description",
                            "Hex head, stainless");
    if (ret != KV_SUCCESS) {
        printf("row put 'description' failed.\n");
        goto cleanup;
    }
    ret = kv_row_put_int(row, "count", 5);
    if (ret != KV_SUCCESS) {
        printf("row put 'count' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_float(row, "percentage", 0.2173913);
    if (ret != KV_SUCCESS) {
        printf("row put 'percentage' failed.\n");
        goto cleanup;
    }

    ret = kv_table_put(store, "myTable", row,
                        NULL); // new version
    if (ret != KV_SUCCESS) {
        printf("Store put failed.\n");
        goto cleanup;
    } else {
        printf("Store put succeeded.\n");
    }

cleanup:
    if (row) {
        kv_release_row(&row);
    }
}
```

Other versions of `kv_table_put()` exist which allow you to provide options and version information, and so forth.

Deleting a Table Row

Use `kv_table_delete()` to delete a table row. Notice that this function does not return `KV_ERROR_T`, but instead returns an integer.

```
void
do_store_ops(kv_store_t *store)
{
    kv_error_t ret;
    kv_row_t *key = NULL;

    key = kv_create_row();
    if (!key) {
       printf("key creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(key, "item", "Bolts");
    if (ret != KV_SUCCESS) {
        printf("row put 'item' failed.\n");
        goto cleanup;
    }

    ret = kv_table_delete(store, "myTable", key);
    // ret is 1 if a row was deleted
    //        0 if no row with the provided key was found
    //        < 0 if an error occurred.
    if (ret < 0) {
        printf("Row deletion failed. %i\n", ret);
        goto cleanup;
    } else {
        printf("Row deletion succeeded.\n");
    }

cleanup:
    if (key) {
        kv_release_row(&key);
    }
}
```

Other versions of `kv_table_delete()` exist which allow you to provide options and version information, and so forth.

Reading a Single Table Row

To read a single table row, create a `kv_row_t` structure set with the field names and field values contained by the row that you want to retrieve. Then create a second `kv_row_t` structure that you will use to hold the retrieved row. The row is retrieved using `kv_table_get()`. You can then examine the various fields in the retrieved row using the proper version of `kv_row_get_xxxx()`, where `xxxx` is the datatype of the field that you are examining.

For example, to retrieve the table row created in Writing to a Table Row:

```
void
do_store_ops(kv_store_t *store)
{
    kv_error_t ret;
    kv_row_t *key = NULL;

    key = kv_create_row();
    if (!key) {
       printf("key creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(key, "item", "Bolts");
    if (ret != KV_SUCCESS) {
        printf("row put 'item' failed.\n");
        goto cleanup;
    }

    kv_row_t *retRow = NULL;
    retRow = kv_create_row();
    if (!retRow) {
       printf("retRow creation failed.\n");
       goto cleanup;
    }
    ret = kv_table_get(store, "myTable", key, &retRow);
    if (!retRow) {
        printf("Row retrieval failed.\n");
        goto cleanup;
    }

    const char *retItem = NULL, *retDescription = NULL;
    int retCount = 0;
    float retPercentage = 0.0;

    kv_row_get_string(retRow, "item", &retItem);
    kv_row_get_string(retRow, "description", &retDescription);
    kv_row_get_int(retRow, "count", &retCount);
    kv_row_get_float(retRow, "percentage", &retPercentage);

    printf("Item: %s. Desc: %s. Count is %i. Percent is %f\n",
            retItem, retDescription, retCount, retPercentage);

cleanup:
    if (key) {
        kv_release_row(&key);
    }

    if (retRow) {
        kv_release_row(&retRow);
    }
}
```

Reading Multiple Table Rows

Use `kv_table_multi_get()` or `kv_table_iterator()` to read multiple rows from a table at a time. These functions require you to provide a `kv_row_t` structure that serves as the lookup key. Different restrictions apply to the key you provide, depending on the function that you use. The example provided here uses `kv_table_multi_get()` which requires that the provided key at least contains all the table's shard keys. If all of the shard keys are not present, then the function will return without error, but without any results.

`kv_table_multi_get()` populates a `kv_iterator_t` structure, which you iterate over using `kv_iterator_next()`. Use `kv_iterator_get_result()` and `kv_result_get_row()` to retrieve the row available for each position in the result set.

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:

```
int main(void) {
    kv_store_t *store = NULL;

    open_store(&store);
    if (!store) {
        goto ERROR;
    }

    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);

    retrieve_table_rows(store);

ERROR:
    /* Close the store handle. */
    if (store) {
        kv_close_store(&store);
    }
    return 0;
}

void
do_store_ops(kv_store_t *store,
             const char *itemType, const char *itemCategory,
             const char *itemClass, const char *itemColor,
             const char *itemSize, float price, int inventoryCount)
{
    kv_error_t ret;
    kv_row_t *row = NULL;

    row = kv_create_row();
    if (!row) {
       printf("row creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(row, "itemType", itemType);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemType' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "itemCategory", itemCategory);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemCategory' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "itemClass", itemClass);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemClass' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "itemColor", itemColor);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemColor' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "itemSize", itemSize);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemSize' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_float(row, "price", price);
    if (ret != KV_SUCCESS) {
        printf("row put 'price' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_int(row, "inventoryCount", inventoryCount);
    if (ret != KV_SUCCESS) {
        printf("row put 'inventoryCount' failed.\n");
        goto cleanup;
    }

    ret = kv_table_put(store, "myTable", row,
                        NULL); // new version
    if (ret != KV_SUCCESS) {
        printf("Store put failed.\n");
        goto cleanup;
    } else {
        printf("Store put succeeded.\n");
    }

cleanup:
    if (row) {
        kv_release_row(&row);
    }
}
```

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 `kv_table_iterator` with an empty row for the `key` parameter.)

```
 void
retrieve_table_rows(kv_store_t *store)
{
    kv_error_t ret;
    kv_iterator_t *iter = NULL;
    kv_row_t *key = NULL;

    key = kv_create_row();
    if (!key) {
       printf("key creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(key, "itemType", "Hats");
    if (ret != KV_SUCCESS) {
        printf("row put 'itemType' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(key, "itemCategory", "baseball");
}

if (ret != KV_SUCCESS) {
    printf("row put 'itemCategory' failed.\n");
    goto cleanup;
}

ret = kv_row_put_string(key, "itemClass", "longbill");
if (ret != KV_SUCCESS) {
    printf("row put 'itemClass' failed.\n");
    goto cleanup;
}

ret = kv_table_multi_get(store,
                         "myTable",
                         key,
                         KV_FALSE /* Keyonly*/,
                         NULL /* Field range */,
                         NULL /* Included tables */,
                         NULL /* Read options */,
                         &iter);

int nRows = 0;
while (kv_iterator_next(iter) == KV_SUCCESS) {
    const kv_result_t *result;
    kv_row_t *retRow;
    const char *itemType = NULL;
    const char *itemCategory = NULL;
    const char *itemClass = NULL;
    const char *itemColor = NULL;
    const char *itemSize = NULL;
    float price = 0.0;
    int inventoryCount = 0;

    result = kv_iterator_get_result(iter);
    retRow = kv_result_get_row(result);

    nRows++;

    kv_row_get_string(retRow, "itemType", &itemType);
    kv_row_get_string(retRow, "itemCategory", &itemCategory);
    kv_row_get_string(retRow, "itemClass", &itemClass);
    kv_row_get_string(retRow, "itemSize", &itemSize);
    kv_row_get_string(retRow, "itemColor", &itemColor);
    kv_row_get_float(retRow, "price", &price);
    kv_row_get_int(retRow, "inventoryCount", &inventoryCount);

    printf("Row %d:\n", nRows);
    printf("\t%s, %s, %s:\n", itemType, itemCategory, itemClass);
    printf("\t\tColor: %s    itemSize: %s\n", itemColor,
            itemSize);
    printf("\t\tprice: %f    inventory: %d\n", price,
            inventoryCount);
}
```

```
cleanup:
    if (key) {
        kv_release_row(&key);
    }

    if (iter) {
        kv_release_iterator(&iter);
    }
}
```

Reading Using Indexes

Use `kv_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 `kv_row_t` structure that represents the indexed field(s) and value(s) that you want retrieved. The second way is to provide a `kv_field_range_t` structure that identifies starting and ending index values that you want returned. The `kv_field_range_t` and `kv_row_t` structures can be used together to restrict the return set values.

If both the `kv_row_t` and `kv_field_range_t` values are `NULL`, 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:

```
int main(void) {
    kv_store_t *store = NULL;

    open_store(&store);
    if (!store) {
        goto ERROR;
    }


    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");

    read_index(store);


ERROR:
    /* Close the store handle. */
    if (store) {
        kv_close_store(&store);
    }
    return 0;
}

void
do_store_ops(kv_store_t *store,
             const char *surname, const char *familiarName,
             const char *userID, const char *phone,
             const char *address, const char *email,
             const char *birthdate)
{
    kv_error_t ret;
    kv_row_t *row = NULL;

    row = kv_create_row();
    if (!row) {
       printf("row creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(row, "surname", surname);
    if (ret != KV_SUCCESS) {
        printf("row put 'surname' failed.\n");
        goto cleanup;
    }
    ret = kv_row_put_string(row, "familiarName", familiarName);
    if (ret != KV_SUCCESS) {
        printf("row put 'familiarName' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "userID", userID);
    if (ret != KV_SUCCESS) {
        printf("row put 'userID' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "phonenumber", phone);
    if (ret != KV_SUCCESS) {
        printf("row put 'phonenumber' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "address", address);
    if (ret != KV_SUCCESS) {
        printf("row put 'address' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "email", email);
    if (ret != KV_SUCCESS) {
        printf("row put 'email' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_string(row, "dateOfBirth", birthdate);
    if (ret != KV_SUCCESS) {
        printf("row put 'birthdate' failed.\n");
        goto cleanup;
    }



    ret = kv_table_put(store, "myTable", row,
                        NULL); // new version
    if (ret != KV_SUCCESS) {
        printf("Store put failed.\n");
        goto cleanup;
    } else {
        printf("Store put succeeded.\n");
    }

cleanup:
    if (row) {
        kv_release_row(&row);
    }
}
```

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 the result set to be empty. Comment both `BLOCK 1` and `BLOCK 2` in order to print the entire table.

```
void
read_index(kv_store_t *store)
{
    kv_error_t ret;
    kv_iterator_t *iter = NULL;
    kv_row_t *key = NULL;
    kv_field_range_t *rangep = NULL;

    key = kv_create_row();
    if (!key) {
       printf("key creation failed.\n");
       goto cleanup;
    }

    // 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 the result set
    // will be empty.
    //
    //ret = kv_row_put_string(key, "dateOfBirth", "1988-02-20");
    //if (ret != KV_SUCCESS) {
    //    printf("row put 'dateOfBirth' failed.\n");
    //    goto cleanup;
    //}

    // 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.
    ret = kv_create_field_range("dateOfBirth", // Field
                                "1990-01-01",  // Start value
                                KV_TRUE,       // Inclusive?
                                "2000-01-01",  // End value
                                KV_TRUE,       // Inclusive?
                                &rangep);
    if (ret != KV_SUCCESS) {
        printf("field range creation failed.\n");
        goto cleanup;
    }

    ret = kv_index_iterator(store, "myTable", "DoB",
                            key, // Key to use for the lookup
                            KV_FALSE, // Whether only primary keys
                                      // are returned
                            rangep, // Field range
                            NULL,  // Included tables
                            KV_DIRECTION_UNORDERED,
                            NULL,  // Read options
                            0, // max iterator results, 0 - use default
                            &iter);
    int nRows = 0;
    while (kv_iterator_next(iter) == KV_SUCCESS) {
        const kv_result_t *result;
        kv_row_t *retRow;
        const char *surname = NULL;
        const char *familiarName = NULL;
        const char *userID = NULL;
        const char *phonenumber = NULL;
        const char *address = NULL;
        const char *email = NULL;
        const char *dateOfBirth = NULL;

        result = kv_iterator_get_result(iter);
        retRow = kv_result_get_row(result);

        nRows++;

        kv_row_get_string(retRow, "surname", &surname);
        kv_row_get_string(retRow, "familiarName", &familiarName);
        kv_row_get_string(retRow, "userID", &userID);
        kv_row_get_string(retRow, "phonenumber", &phonenumber);
        kv_row_get_string(retRow, "address", &address);
        kv_row_get_string(retRow, "email", &email);
        kv_row_get_string(retRow, "dateOfBirth", &dateOfBirth);

        printf("Row %d:\n", nRows);
        printf("\t%s, %s (%s):\n", familiarName, surname, userID);
        printf("\t\tPhone: %s\n", phonenumber);
        printf("\t\tEmail: %s\n", email);
        printf("\t\tAddress: %s\n", address);
        printf("\t\tDoB: %s\n", dateOfBirth);
    }

cleanup:
    if (key) {
        kv_release_row(&key);
    }

    if (rangep) {
        kv_release_field_range(&rangep);
    }

    if (iter) {
        kv_release_iterator(&iter);
    }
}
```

Sequence Execution

Use `kv_create_operations()` to create a `kv_operations_t` structure to hold a sequence of write operations. All the write operations will execute as a single atomic structure so long as all the operations share the same shard key.

You populate operations to the `kv_operations_t` structure using a series of one or more `kv_create_table_xxxx_op()` functions, where `xxxx` indicates the type of operation to insert into the operations list. For example, `kv_create_table_delete_op()` inserts a row deletion operation into the sequence.

The operations sequence is executed using `kv_table_execute_operations()`.

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

```
int main(void) {
    kv_error_t ret;
    kv_store_t *store = NULL;
    kv_operations_t *op = NULL;
    kv_operation_results_t *resultp = NULL;

    open_store(&store);
    if (!store) {
        goto ERROR;
    }


    op = kv_create_operations();

    // Causes all rows to be released when op is released.
    kv_operations_set_donate(op);

    ret = add_op_list(op, "Hats", "baseball", "longbill",
                 "red", "small", 13.07, 107);
    if (ret != KV_SUCCESS) {
        printf("adding to op list failed.\n");
        return -1;
    }

    ret = add_op_list(op, "Hats", "baseball", "longbill",
                 "red", "medium", 14.07, 198);
    if (ret != KV_SUCCESS) {
        printf("adding to op list failed.\n");
        return -1;
    }

    ret = add_op_list(op, "Hats", "baseball", "longbill",
                 "red", "large", 15.07, 140);
    if (ret != KV_SUCCESS) {
        printf("adding to op list failed.\n");
        return -1;
    }

    ret = kv_table_execute_operations(store, op, NULL, &resultp);
    if (ret != KV_SUCCESS) {
        printf("operation execution failed.\n");
    } else {
        printf("operation execution succeeded.\n");
    }

ERROR:
    /* Close the store handle. */
    if (store) {
        kv_close_store(&store);
    }

    if (resultp) {
        kv_release_operation_results(&resultp);
    }

    if (op) {
        kv_release_operations(&op);
    }

    return 0;
}

int
add_op_list(kv_operations_t *op,
            const char *itemType, const char *itemCategory,
            const char *itemClass, const char *itemColor,
            const char *itemSize, float price, int inventoryCount)
{
    kv_error_t ret;
    kv_row_t *row = NULL;

    row = kv_create_row();
    if (!row) {
        printf("row creation failed.\n");
        return -1;
    }

    ret = kv_row_put_string(row, "itemType", itemType);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemType' failed.\n");
        return -1;
    }

    ret = kv_row_put_string(row, "itemCategory", itemCategory);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemCategory' failed.\n");
        return -1;
    }

    ret = kv_row_put_string(row, "itemClass", itemClass);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemClass' failed.\n");
        return -1;
    }

    ret = kv_row_put_string(row, "itemColor", itemColor);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemColor' failed.\n");
        return -1;
    }

    ret = kv_row_put_string(row, "itemSize", itemSize);
    if (ret != KV_SUCCESS) {
        printf("row put 'itemSize' failed.\n");
        return -1;
    }

    ret = kv_row_put_float(row, "price", price);
    if (ret != KV_SUCCESS) {
        printf("row put 'price' failed.\n");
        return -1;
    }

    ret = kv_row_put_int(row, "inventoryCount", inventoryCount);
    if (ret != KV_SUCCESS) {
        printf("row put 'inventoryCount' failed.\n");
        return -1;
    }

    ret = kv_create_table_put_op(op, "myTable", row,
                  KV_RETURN_ROW_NONE, // kv_return_row_version_enum
                  0); // Abort on failure?
    if (ret != KV_SUCCESS) {
        printf("Store put op failed.\n");
        return -1;
    } else {
        printf("Store put op succeeded.\n");
    }

    // Do not release the row at the end of this, as doing so will
    // cause the operation execution to core dump. The row must be
    // saved for the future operation.
    //
    return KV_SUCCESS;
}
```

Setting Consistency Guarantees

By default, read operations are performed with a consistency of guarantee of KV_CONSISTENCY_NONE. Use one of the following functions to create a consistency guarantee that overrides this default:

1. kv_create_simple_consistency()
2. kv_create_time_consistency()
3. kv_create_version_consistency()

These allocate and populate a kv_consistency_t structure that must be released using kv_release_consistency().

You then use the kv_consistency_t structure with kv_create_read_options() to create allocate and populate a kv_read_options_t structure. Use kv_release_read_options() to release this structure.

Finally, use the kv_read_options_t structure when performing a read operation from the store.

For example, the code fragment shown in Reading a Single Table Row can be rewritten to use a default consistency policy in the following way:

```
void
do_store_ops(kv_store_t *store)
{
    kv_consistency_t *consis = NULL;
    kv_error_t ret;
    kv_read_options_t *readopts = NULL;
    kv_result_t *results = NULL;
    kv_row_t *key = NULL;

    ret = kv_create_simple_consistency(KV_CONSISTENCY_ABSOLUTE,
                                       &consis);
    if (ret != KV_SUCCESS) {
        printf("consistency creation failed\n");
        goto cleanup;
    }

    ret = kv_create_read_options(consis,  // consistency
                                 0,       // timeout value.
                                          // 0 means use the default.
                                 &readopts);
    if (ret != KV_SUCCESS) {
        printf("readoptions creation failed\n");
        return;
    }

    key = kv_create_row();
    if (!key) {
       printf("key creation failed.\n");
```

```
        goto cleanup;
    }

    ret = kv_row_put_string(key, "item", "Bolts");
    if (ret != KV_SUCCESS) {
        printf("row put 'item' failed.\n");
        goto cleanup;
    }

    ret = kv_table_get_with_options(store,
                                    "myTable",
                                    key,
                                    readopts,
                                    &results);
    if (ret != KV_SUCCESS) {
        printf("Retrieval failed.\n");
        goto cleanup;
    }

    kv_row_t *retRow = kv_result_get_row(results);

    const char *retItem = NULL, *retDescription = NULL;
    int retCount = 0;
    float retPercentage = 0.0;

    kv_row_get_string(retRow, "item", &retItem);
    kv_row_get_string(retRow, "description", &retDescription);
    kv_row_get_int(retRow, "count", &retCount);
    kv_row_get_float(retRow, "percentage", &retPercentage);

    printf("Item: %s. Desc: %s. Count is %i. Percent is %f\n",
            retItem, retDescription, retCount, retPercentage);

cleanup:
    if (key) {
        kv_release_row(&key);
    }

    if (retRow) {
        kv_release_row(&retRow);
    }

    // kv_release_read_options also releases the
    // kv_consistency_t structure.
    if (readopts) {
        kv_release_read_options(&readopts);
    }
}
```

Setting Durability Guarantees

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

Use kv_create_durability() to initialize a kv_durability_t structure. You then use the kv_durability_t structure with kv_create_write_options() to allocate and populate a kv_write_options_t structure. Use kv_release_write_options() to release this structure.

Finally, use the kv_write_options_t structure when performing a write operation in the store.

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

```
void
do_store_ops(kv_store_t *store)
{
    kv_error_t ret;
    kv_row_t *row = NULL;
    kv_durability_t durability;
    kv_write_options_t *writeopts;
    kv_result_t *results;

    row = kv_create_row();
    if (!row) {
       printf("row creation failed.\n");
       goto cleanup;
    }

    ret = kv_row_put_string(row, "item", "Bolts");
    if (ret != KV_SUCCESS) {
        printf("row put 'item' failed.\n");
        goto cleanup;
    }
    ret = kv_row_put_string(row, "description", "Hex head, stainless");
    if (ret != KV_SUCCESS) {
        printf("row put 'description' failed.\n");
        goto cleanup;
    }
    ret = kv_row_put_int(row, "count", 5);
    if (ret != KV_SUCCESS) {
        printf("row put 'count' failed.\n");
        goto cleanup;
    }

    ret = kv_row_put_float(row, "percentage", 0.2173913);
    if (ret != KV_SUCCESS) {
        printf("row put 'percentage' failed.\n");
        goto cleanup;
    }

    durability = kv_create_durability(
                    KV_SYNC_FLUSH, // Master sync
                    KV_SYNC_NONE,  // Replica sync
                    KV_ACK_MAJORITY); // Ack policy

    ret = kv_create_write_options(durability,
                                  0, // 0 is default timeout
                                  &writeopts);
    if (ret != KV_SUCCESS) {
        printf("Write options creation failed.\n");
        return;
    }


    ret = kv_table_put_with_options(store,
                                    "myTable",
                                    row,
                                    writeopts,
                                    // Whether the new row should be
                                    // returned in the results
                                    // parameter.
                                    KV_RETURN_ROW_NONE,
                                    &results);

    if (ret != KV_SUCCESS) {
        printf("Store put failed.\n");
        goto cleanup;
    } else {
        printf("Store put succeeded.\n");
    }

cleanup:
    if (row) {
        kv_release_row(&row);
    }

    if (writeopts) {
        kv_release_write_options(&writeopts);
    }

    if (results) {
        kv_release_result(&results);
        }
}
```

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.

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,

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

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.

Oracle® NoSQL Database C Table Driver Developer's Guide, Release 20.2
E85383-09

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.