AS101 Lab Exercises
Python

Objective

After successful completion of this Lab module you will have:

- Connected to a cluster
- Write and read records using simple and complex values
- Used advanced key-value techniques

The exercises can be cloned or downloaded at:

- https://github.com/aerospike-edu/student-workbook.git

```
$cd ~
$git clone https://github.com/aerospike-edu/student-workbook.git
```

Lab Overview

The lab exercises add functionality to a simple Twitter- like console application (tweetaspike) using Aerospike as the database.

In this Lab, we will focus on key-value operations and techniques. You will add code to:
- Create users and Tweets
- Read all the Tweets for a user
- Use an advance Key-value feature

The exercises shell is located in your cloned GitHub directory
```
~/exercises/Key-valueOperations/<language>
```
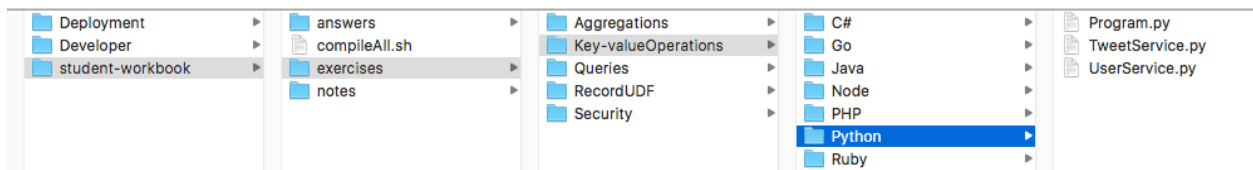
Make sure you have your server up and you know its IP address

In your cloned or downloaded repository, you will find the following directories:
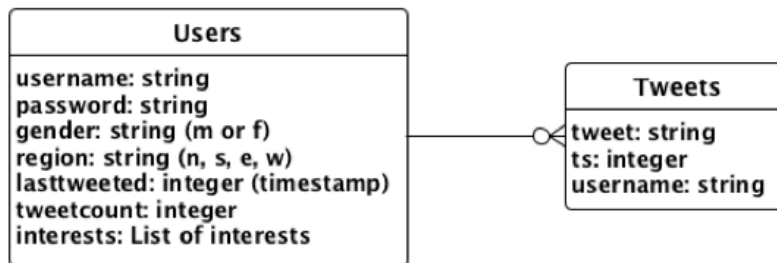- answers
- exercises
- notes

In the exercises subdirectory, select the Key-value Operations subdirectory and then within it, the Python subdirectory. Your task is to complete the code as outlined in each exercise.

Make sure you have the Aerospike Server up and you know its IP address.

Deployment	answers	Aggregations	C#	Program.py
Developer	compileAll.sh	Key-valueOperations	Go	TweetService.py
student-workbook	exercises	Queries	Java	UserService.py
	notes	RecordUDF	Node	
		Security	PHP	
			Python	
			Ruby	

Tweetaspike Data Model

- In the lab exercises, you will build a twitter like module and focus on how you will query the data. For example:
  - All Tweets from a given user
  - Give me the last 10 Tweets for a given user
  - Last 10 Tweets for all users
  - How many users Tweeted in the last X minutes

Users

username: string
password: string
gender: string (m or f)
region: string (n, s, e, w)
lasttweeted: integer (timestamp)
tweetcount: integer
interests: List of interests

Tweets

tweet: string
ts: integer
username: string

Users

Namespace: test, Set: users, Key:<username>

Bin Name	Type	Comment
username	String	
password	String	For simplicity password is stored in plain text
gender	String	'm' or 'f'
region	String	Valid values are: 'n' (North), 's','e', 'w' to keep data entry to minimal.
lasttweeted	Integer	Stores epoch timestamp of the last/most recent tweet. Default to 0.
tweetcount	Integer	Stores total number of tweets for the user. Defautl 0.
interests	List	A list of interests

Tweets

Namespace: test, Set: tweets, Key:<username:<counter>>

Bin Name	Type	Comment
tweet	String	Tweet text
ts	Integer	Stores epoch timestamp of tweet
username	String	User name of tweeter.

Python Lab Exercises

Python Client Install

- Aerospike Python Client uses Aerospike C Client. Depending on your machine, follow installation instructions at:
    - http://www.aerospike.com/docs/client/python/install
- If using Vagrant with Virtual Box, go to ClientInstall sub-directory and run the shell script:
    - `$./as_python_install.sh`
- `Redhat/Centos:`
    - `sudo yum install python-devel`
    - `sudo yum install openssl-devel`
    - `sudo yum install python-pip`
    - `sudo pip install aerospike`
- `Debian/Ubuntu:`
    - `sudo apt-get install python-dev`
    - `sudo apt-get install libssl-dev`
    - `sudo apt-get install python-pip`
    - `sudo pip install aerospike`
- When doing the exercises, refer to online python client documentation for examples.
    - http://www.aerospike.com/docs/client/python/usage

For example, on Ubuntu:
```
$sudo apt-get install python-dev
$sudo apt-get install libssl-dev
$sudo apt-get install python-pip
```
Then,
```
$sudo pip install aerospike
```
If older version already exists, do:
```
$sudo pip install --upgrade aerospike
```

Note: If python client is not installed, when you run the lab exercises, you will see error in the `import aerospike` call.

Aerospike python library is installed in [python shell, import aerospike, print(aerospike)]:
```
/usr/local/lib/python2.7/dist-packages/
aerospike-2.0.x.egg-info
aerospike.pth
aerospike.so
```

Key-Value Operations Exercise - Goals

This module describes how to use Key-value operations. At the end of this module you will be able to code in Python to:

- Connect and Disconnect from an Aerospike cluster
- Perform Write operations
- Perform Read operations
- Apply advanced key-value techniques
- Correctly handle errors

Exercise 1 – Python: Connect & Disconnect

Open `Program.py` for edit.

- Create a client instance with one initial IP address. Ensure connection is created only once.
- Add code to close the connection and disconnect from the cluster. Ensure this code is executed only once.

Work in `..student-workbook-master/exercises/Key-valueOperations/Python`
It has three files: `Program.py, TweetService.py` and `UserService.py`

Refer to Python client documentation at
http://www.aerospike.com/docs/clients/python/usage

Search for "Exercise" or "TODO" in comments in the code. It currently opens connection to Aerospike server on localhost.

Modify it to open connection on your Aerospike Server running on AWS. Can you modify line 59 to change the default IP address? In case of a cluster, can you add multiple server IP addresses on line 46?

`Line 46, Program.py` -> Add try/except around connect() on line 46. Add code to except if connection was not opened successfully.

```
def __init__(self, host, port, namespace, set):
    # TODO: Establish a connection to Aerospike cluster
    # Exercise 1
    print("\nTODO: Establish a connection to Aerospike cluster");
    # TODO: Check to see if the cluster connection succeeded
    try:
        self.client = aerospike.client({ 'hosts': [ (host, port) ] }).connect()
    except ClientError as e:
        print("\nERROR: Connection to Aerospike cluster failed!")
        print("\nPlease check the server settings and try again!")
        print("Error: {0} [{1}]".format(e.msg, e.code))
        sys.exit(1)
    self.seedHost = host
    self port = port
```

Line 79, Program.py TODO-> ignore, handled in Line 46 above.

Line 136, Program.py : The program currently does not exit gracefully () – it does not close the connection to the server. Add the call to close the connection to the server.

```
            ts.createTweets()
        else:
            print ("Enter a Valid number from above menue !!")
    # TODO: Close Aerospike cluster connection
    # Exercise 1
    print("\nClosed Aerospike cluster connection");
    self.client.close()
```

Exercise 2 – Python: Write & Read Records

Create a User Record and Tweet Record.
Locate UserService and TweetService classes.

1 – Create a User Record – in UserService.createUser()
- Create an instance of WritePolicy (Optional)
- Create Key and Bin instances for the User Record
- Write User Record

2 – Create a Tweet Record – in TweetService.createTweet()
- Prior to writing, validate user exists.
- Read User Record. If valid user:
- Create an instance of WritePolicy ("exists":CREATE)
- Create Key and Bin instances for the Tweet Record
- Write Tweet Record
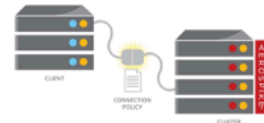
Start with UserService.py
Use python string utility function split(delimiter) to break the csv string into a python list.

```python
record = { "username": username }
if len(username) > 0:
    #  Get password
    record['password'] = raw_input("Enter password for " + username + ":")
    #  Get gender
    record['gender'] = raw_input("Select gender (f or m) for " + username + ":")
    #  Get region
    record['region'] = raw_input("Select region (north, south, east or west) for " + username + ":")
    #  Get interests
    record['interests'] = raw_input("Enter comma-separated interests for " + username + ":")
    # TODO: Create Key and Bin(s) for the user record. Remember to convert
    # comma-separated interests into a list before storing it.
    # Exercise 2
    print("\nTODO: Create Key and Bin instances for the user record.")
    print("\nRemember to convert comma-separated interests into a list before storing it.");

    record['interests']=record['interests'].split(",")
    reckey = ('test', 'users', record['username'])

    #TODO: Write user record
    #Exercise 2
    print("\nTODO: Write user record");

    self.client.put(reckey, record)
```

```python
def getUser(self):
    userRecord = None
    userKey = None
    #  Get username
    username = str()
    username = raw_input("Enter username: ")
    if len(username) > 0:
        #  Check if username exists
        # TODO: Read user record
        # Exercise 2
        reckey = ('test', 'users', username)
        (reckey, meta, userRecord) = self.client.get(reckey)
        if userRecord:
            print("\nINFO: User record read successfully! Here are the details:\n")
            # TODO: Output user record to the console. Remember to convert
            # comma-separated interests into a list before outputting it
            # Exercise 2
            print(userRecord)
```

Edit TweetService.py to implement write policy – create tweet only if it does not exist. Policy key should be 'exist', Policy Value should be POLICY_EXISTS_CREATE.

```python
reckey = ('test', 'users', username)
(reckey, meta, userRecord) = self.client.get(reckey)
if userRecord:
    # Set Tweet Count
    if 'tweetcount' in userRecord:
        nextTweetCount = int(userRecord['tweetcount']) + 1
    else:
        nextTweetCount = 1
    #  Get tweet
    record['tweet'] = raw_input("Enter tweet for " + username + ":")
    #  Create timestamp to store along with the tweet so we can
    #  query, index and report on it
    ts= self.getTimeStamp()
    # TODO: Create WritePolicy instance
    # Exercise 2
    print("\nTODO: Create WritePolicy instance");

    #Set the 'exists' policy ie what to do if the record exists
    #to POLICY_EXISTS_CREATE - create record only if it does not exist
    policy = {'exists': aerospike.POLICY_EXISTS_CREATE}

    #TODO: Create Key and Bin instances for the tweet record.
    #HINT: tweet key should be in username:nextTweetCount format
    # Exercise 2
    print("\nTODO: Create Key and Bin instances for the tweet record");
    tweetKey = ('test', 'tweets', username + ':'+ str(nextTweetCount))
    record['ts']=ts
    record['username']=username
    # TODO: Write tweet record
    # Exercise 2
    print("\nTODO: Write tweet record");
    self.client.put(tweetKey, record, policy)
    # TODO: Update tweet count and last tweeted timestamp in the user
    # Exercise 2
    # We are updating an existing record - UPDATE is default write policy
    userKey = ('test','users',username)
    userRecord['lasttweeted']=ts
    userRecord['tweetcount']=nextTweetCount
    self.client.put(userKey, userRecord)
    print("\nINFO: Tweet record created!\n",record,tweetKey)
    #  Update tweet count and last tweet'd timestamp in the user record
else:
    print("ERROR: User record not found!\n")
```

Finally, in TweetService.py, add code to updateUser().

```python
def updateUser(self, client, userKey, policy, ts, tweetCount):
    # TODO: Update tweet count and last tweeted timestamp in the user record
    # Exercise 2
    print("\nTODO: Update tweet count and last tweeted timestamp in the user record")
    #userKey = ('test','users',username)
    userRecord['lasttweeted']=ts
    userRecord['tweetcount']=tweetCount
    client.put(userKey, userRecord)
```

Explore all other policy settings in API documentation.

Exercise 3 – Python: Batch Reads

Batch read all tweets by a given user. Use get_many() to read many Records in one operation.

- Pass Array of Keys
- Records – List of tuples of (key, meta, bins) returned in the same order as the Keys
- In TweetService.py, write code in batchGetUserTweets()

In UserService.py, add code to read all tweets by a user in batchGetUserTweets() function. get_many() API call will return an Array of Tuples. Each Tuple contains (key, meta, bins). Using bins dictionary at tuple index 2, print the 'tweet'.

```python
def batchGetUserTweets(self):
    userRecord = None
    userKey = None
    #  Get username
    username = str()
    username = raw_input("Enter username: ")
    if len(username) > 0:
        #  Check if username exists
        # TODO: Read user record
        # Exercise 3

        reckey = ('test', 'users', username)
        (reckey, meta, userRecord) = self.client.get(reckey)

        print("\nTODO: Read user record")
        if userRecord:
            # TODO: Get how many tweets the user has
            # Exercise 3
            print("\nTODO: Get how many tweets the user has")
            tweetCount = userRecord['tweetcount']
            # TODO: Create an array of tweet keys -- keys[tweetCount]
            # Exercise 3
            keys = []
            pkey = []
            for i in range(1, tweetCount+1):
                pkey.append(userRecord['username']+':'+str(i))
                keys.append( ('test', 'tweets', pkey[i-1]))
            print("\nTODO: Create an array of Key instances -- keys[tweetCount]")
            # TODO: Initiate batch read operation
            # Exercise 3
            print("\nTODO: Initiate batch read operation");
            records = self.client.get_many(keys)
            #Note: get_many() API returns a list of tuples (key, meta, bins)
            # TODO: Output tweets to the console
            # Exercise 3
            print("\nTODO: Output tweets to the console");
            for i in range(1, tweetCount+1):
                #print(records[i-1]) #Prints the full tuple (key, meta, bins)
                print(records[i-1][2]['tweet'])

    else:
        print("ERROR: User record not found!\n")
```

Exercise 4 – Python: Scan All – Get All Tweets by All Users

The **Scan** operation returns all Records in a Set or if None, all records in the namespace.

- foreach() invokes a callback for each record returned.
- result() provides the records from the scan as a List.
- Bins returned can be filtered using select().
    - scan = client.scan('ns','setname')
    - scan.select('bin1name', 'bin2name') ... filters records for results() or foreach()
    - res = scan.results()
    - scan. foreach(mycallbackfunction, options=myScanOptions)
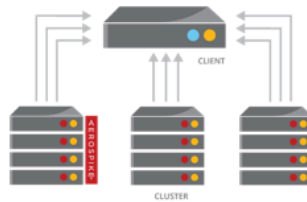
Set scan policy and its parameters (optional), perform a scan operation and process the results.

```python
#printTweet is a callback for Exercise 4
#that takes arg = record tuple of (key, meta, bins)
def printTweet(self, (key,meta,bins)):
    print(bins['username']+" tweeted: "+ bins['tweet'])

def scanAllTweetsForAllUsers(self):
    # Initiate scan operation that invokes callback
    # for outputting tweets on the console
    # Exercise 4
    print("\nTODO: Initiate scan operation that invokes")
    print("callback for outputting tweets to the console\n")
    scan = self.client.scan('test', 'tweets')
    scan.foreach(self.printTweet)
```

Exercise 5 – Python: Read Modify Write (CAS)

Update the User record with a new password ONLY if the User record is un-modified.
This involves, in the UserService.updatePasswordUsingCAS():

- Read the Record and get its generation value.
- Create a Write Policy
- Set WritePolicy.generation to the value read from the User record.
- Set WritePolicy.generationPolicy to EXPECT_GEN_EQUAL
- Modifying the Record at the application.
- Write the modified data with the generation previous Read.
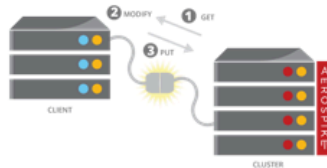
Note: Ensure you Import the exception error definitions:

After,

```
import aerospike
```

Add,

```
from aerospike.exception import *
```

```python
#   Check if username exists
meta = None
policy = None
userKey = ("test", "users", username)
(key, metadata,userRecord) = self.client.get(userKey,policy) |
if userRecord:
    record = {}
    #   Get new password
    record["password"] = raw_input("Enter new password for " + username + ":")
    # TODO: Update User record with new password
    # Exercise 5

    usergen = metadata['gen']
    #print("usergen ="+str(usergen))
    policy = {'gen':aerospike.POLICY_GEN_EQ}
    meta = {'gen':usergen}
    try:
        self.client.put(userKey,record,meta,policy)
    except RecordGenerationError:
        print("put() failed due to generation policy mismatch")
    except AerospikeError as e:
        print("Error: {0} [{1}]".format(e.msg, e.code))

else:
    print("ERROR: User record not found!")
```

Exercise 6 – Python: Use Operate() to set & get tweetcount

Operate() enables multiple operations atomically on the same Record.

Comment out code written in Exercise 2 and,

- InTweetService, modify createTweet() to call updateUser()

- Modify updateUser() to call updateUserUsingOperate()

- Add code in updateUserUsingOperate() to put timestamp, updated tweetcount and then read back the updated record and print the read back tweetcount. Stack all operations using Operate().
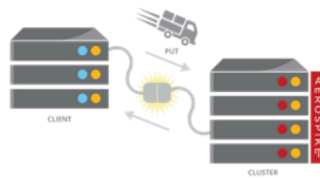
```python
        userRecord[ tweetcount ]=nextTweetCount
        #self.client.put(userKey, userRecord)
        #print("\nINFO: Tweet record created!\n",record,tweetKey)
        #  Update tweet count and last tweet'd timestamp in the user record
        policy = {'timeout': 300}
        self.updateUser(self.client, userKey, policy, ts, nextTweetCount)

    else:
        print("ERROR: User record not found!\n")

def updateUser(self, client, userKey, policy, ts, tweetCount):
    # TODO: Update tweet count and last tweeted timestamp in the user record
    # Exercise 2
    print("\nTODO: Update tweet count and last tweeted timestamp in the user record")
    #userKey = ('test','users',username)
    #userRecord['lasttweeted']=ts
    #userRecord['tweetcount']=tweetCount
    #client.put(userKey, userRecord)
    self.updateUserUsingOperate(client, userKey, policy, ts, tweetCount)


def updateUserUsingOperate(self, client, userKey, policy, ts, tweetCount):
    """ operate now supported in Python Client """
    ops= [{
        "op" : aerospike.OPERATOR_WRITE,
        "bin": "lasttweeted",
        "val": ts
    },
    {
        "op" : aerospike.OPERATOR_WRITE,
        "bin": "tweetcount",
        "val": tweetCount
    },
    {
        "op" : aerospike.OPERATOR_READ,
        "bin": "tweetcount"
    }]
    meta = {}
    (key, meta, bins) = self.client.operate(userKey, ops, meta, policy)

    print("\nOperate(): The tweet count now is: " + str(bins['tweetcount']))
```

Lab: User Defined Functions – Record UDFs

If you want to get familiar with Lua, http://lua-users.org/wiki/TutorialDirectory is a good resource.

Helpful Lua tidbits:

Comments start with - - for eg: `-- This is a comment`
String concatenation operator is .. for eg: `"Hello" .. " World!"`

Tables in Lua are key-value pairs. When key is not supplied, it defaults to int starting with 1.
for eg: `t = {"a", "b", [123]="foo", "c", name="bar", "d", "e"}`, key for "a" is 1, for "c" is 3.

All functions in Lua are a value. ie function is a type in Lua. Function names can be passed as arguments to another function.

for eg: `fname = function(args) body end` OR `function fname(args) body end`
are equivalent declarations.

Note: A call `v:name(args)` is syntactic sugar for `v.name(v,args)`, except that `v` is evaluated only once. This construct is often used in piping streams when writing stream UDFs.

Install Lua in your environment only if you want to play with learning Lua.
https://www.lua.org/download.html and follow directions on the site to make the executable. Depending on your OS, you may need additional libraries. For example, on Ubuntu you need to

```
$sudo apt-get install libreadline-dev
```

`$make linux test` in the `lua-x.x.x` subdirectory will build the `lua` executable in the `src` subdirectory. Running the `lua` executable opens the interactive prompt. Ctrl-C kills it.

To execute any Lua file, for eg: `myprog.lua` containing: `print("Hello".." World!")`

```
$lua myprog.lua
Hello World!
```

Lua overview

Factor example – demonstrating language constructs

```lua
01:function get_all_factors(number)                    -- function definition
02:  local factors = {}                                -- local variable definition
03:  for possible_factor=1, math.sqrt(number), 1 do    -- for loop
04:    local remainder = number%possible_factor        -- assignment (%
05:    if remainder == 0 then                           -- if equal test
06:      local factor, factor_pair
07:           = possible_factor, number/possible_factor -- multiple assignment
08:      table.insert(factors, factor)                  -- table insert
09:      if factor ~= factor_pair then                  -- if not equal
10:        table.insert(factors, factor_pair)
11:      end
12:    end
13:  end
14:  table.sort(factors)
15:  return factors                                     -- return value
16:end
17:-- The Meaning of the Universe is 42.
18:-- Let's find all of the factors driving the Universe.
19:the_universe = 42                                    -- global variable and assignment
20:factors_of_the_universe = get_all_factors(the_universe) -- function call
21:-- Print out each factor
22:print("Count",  "The Factors of Life, the Universe, and Everything")
23:table.foreach(factors_of_the_universe, print)
```

Lua Restrictions in Aerospike UDFs
- No Globals (locals only)
- No Co-routines (no threads)
- No Debug Module (have to use "printf like debugging" via UDF log)
- No call to os.exit() – do not terminate the server process on the node!

Exercise 1 – Write a Record UDF

Locate updateUserPwd.lua file in the udf subfolder and add lua code to:

```
function updatePassword(topRec,pwd)
    -- Exercise 1
    -- TODO: Log current password
    -- TODO: Assign new password to the user record
    -- TODO: Update user record
    -- TODO: Log new password
    -- TODO: return new password
end
```

- Log using debug() function in Aerospike lua API
  - See http://www.aerospike.com/docs/udf/api/logging.html
  - eg: debug("My id: %s", tostring(myIntId))
  - or: debug("My id: "..tostring(myIntId))
- First argument of Aerospike Record UDF must always be of type aeropsike.record which is table of bins and metadata available via its methods.
- Aerospike lua built in methods can be used to manage records.
- eg: aerospike:create(record), aerospike:update(record) etc.

In this exercise, we will write a UDF, register it on the server with AQL and then run it via AQL. We will also setup the UDF log file to see the debug messages.

First, update `/etc/aerospike/aerospike.conf` to enable debug logging of UDF messages.

```
$sudo vi /etc/aerospike/aerospike.conf
```

```
logging {
#       console {
        file /var/log/aerospike/aerospike.log {
                context any info
        }
        file /var/log/aerospike/udf_debug.log {
                context udf debug
        }
}
```

You must restart the aerospike server for the logging update to take effect.
```
$sudo service aerospike restart
```

Run the program to add a user with an initial password.
`$python Program.py` | Create a user.

`cd` to the `udf` sub-directory where you have saved your `updateUserPwd.lua` module.
Start `aql` and register this module.
`$aql`

```
aql> select * from test.users
+----------+--------+--------+-------------------+------------------------+------------+---------------+
| username | gender | region | password          | interests              | tweetcount | lasttweeted   |
+----------+--------+--------+-------------------+------------------------+------------+---------------+
| "pg"     | "m"    | "w"    | "initialPassword" | LIST('["a", "b", "c"]')| 1          | 1472680616296 |
+----------+--------+--------+-------------------+------------------------+------------+---------------+
1 row in set (0.025 secs)
```

Register and Execute UDF via AQL

```
aql> register module './updateUserPwd.lua'
OK, 1 module added.

aql> show modules
+------------------------------------+-------------------+-------+
| hash                               | module            | type  |
+------------------------------------+-------------------+-------+
| "ad4c14c70a7cdd9ea83611dd6fc2621e852cb7b7" | "updateUserPwd.lua" | "lua" |
+------------------------------------+-------------------+-------+
1 row in set (0.000 secs)

aql> execute updateUserPwd.updatePassword('passByUDF') on test.users where PK = "pg"
+----------------+
| updatePassword |
+----------------+
| "passByUDF"    |
+----------------+
```

Check if user password was updated:
```
aql> select * from test.users
+----------+--------+--------+-------------+------------------------+------------+---------------+
| username | gender | region | password    | interests              | tweetcount | lasttweeted   |
+----------+--------+--------+-------------+------------------------+------------+---------------+
| "pg"     | "m"    | "w"    | "passByUDF" | LIST('["a", "b", "c"]')| 1          | 1472680616296 |
+----------+--------+--------+-------------+------------------------+------------+---------------+
1 row in set (0.024 secs)
```

The record UDF file is:

```lua
function updatePassword(topRec,pwd)
    -- Exercise 1
    -- TODO: Log current password
    debug("Current Password is: "..topRec['password'])
    -- TODO: Assign new password to the user record
    topRec['password'] = pwd
    -- TODO: Update user record
    aerospike:update(topRec)
    -- TODO: Log new password
    debug("New Password is: %s", topRec['password'])
    -- TODO: return new password
    return topRec['password']
end
```

For debugging, use grep to search through the udf_debug.log file.

```
$grep "New Password is" /var/log/aerospike/udf_debug.log
```

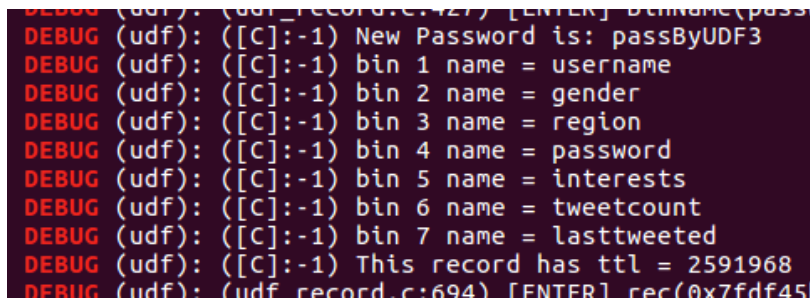**Note:** Sometimes grep may fail to print if it finds binary data in the log file. In that case, try:

```
$cat -v /var/log/aerospike/udf_debug.log | grep "New Password is"
```

You can further explore the Aerospike `record` API in the above module by adding:

```lua
-- Bonus exercises!
-- get bin names
t = record.bin_names(topRec)
for i, name in ipairs(t) do
  debug("bin %d name = %s", i, tostring(name))
end
-- get record ttl
recttl = record.ttl(topRec)
debug("This record has ttl = %d", recttl)
```

and exploring the output in the log file (`/var/log/aerospike/udf_debug.log`)

```
DEBUG (udf): ([C]:-1) New Password is: passByUDF3
DEBUG (udf): ([C]:-1) bin 1 name = username
DEBUG (udf): ([C]:-1) bin 2 name = gender
DEBUG (udf): ([C]:-1) bin 3 name = region
DEBUG (udf): ([C]:-1) bin 4 name = password
DEBUG (udf): ([C]:-1) bin 5 name = interests
DEBUG (udf): ([C]:-1) bin 6 name = tweetcount
DEBUG (udf): ([C]:-1) bin 7 name = lasttweeted
DEBUG (udf): ([C]:-1) This record has ttl = 2591968
DEBUG (udf): (udf_record.c:694) [ENTER] rec(0x7fdf45
```

Note on Registering UDFs –Python Client

While Record UDFs run only on the server, stream UDFs run both on server and the client. Both server and client nodes also have Aerospike system Lua modules, server side installed during server installation on server's `/opt/aerospike/sys/udf/lua` directory, client side aerospike Lua modules (not a copy of server side aerospike Lua modules) installed when installing the client at `/usr/local/aerospike/lua` on the client node.

When user registers user UDF in Lua via AQL, it is only added to the server side, user UDF directory `/opt/aerospike/usr/udf/lua`. Depending on the client, for stream UDFs, user must also upload user UDFs to `/usr/local/aerospike/usr-lua` on client node. If using the client api `udf_put()`, ensure client has write privileges on `/usr/local/aerospike/usr-lua`.

You can do: `$sudo chown <user-id> /usr/local/aerospike/usr-lua`

In Python client, when you use `udf_put()` in application code, it will upload the user UDF both on server and client side, provided you have passed the user side configuration when connecting to the client via the `config` parameter. For example, in client application do:

```python
config = {'hosts': [('127.0.0.1', 3000)],
                'lua': {'system_path':'/usr/local/aerospike/lua/',
                'user_path':'/usr/local/aerospike/usr-lua/'}}
   self.client = aerospike.client(config).connect()
```

Exercise 2: Python – Register a Lua module, Apply UDF

In Python, unlike other clients, you "**apply**" instead of "**execute**" a UDF:

- In Program.py, configure client connection to specify local lua user and system paths. (Optional)
- In UserService.py edit updatePasswordUsingUDF()
  - Register the UDF
    - `client.udf_put('/path.../modulename.lua')`
    - **Observe**: A copy of udf file is put in lua "user_path" locally.
  - Apply the UDF
    - `client.apply(key_tuple, "modulename", "function_name", [function_args])`
    - `function_args` must be passed as a list
  - Output updated password to console
    - UDF return value is in `client.apply()`

Module: *updateUserPwd.lua*

function **updatePassword**(topRec,pwd)

We will now use the Python application itself to register and execute the UDF instead of `aql` in Exercise 1.

To begin with, remove the module from Exercise 1 using `aql`

```
aql> remove module 'updateUserPwd.lua'
OK, 1 module removed.

aql> show modules
0 rows in set (0.000 secs)
```

Run the python application to create a user with an initial password. Validate with aql using
`aql>select * from test.users`

Modify Program.py at add lua sys and user_path as discussed earlier. (Optional for Record UDFs, required for Stream UDFs later).

Update the code as shown below:

```python
def updatePasswordUsingUDF(self):
    userRecord = None
    userKey = None
    #  Get username
    username = str()
    username = raw_input("Enter username: ")
    if len(username) > 0:
        #  Check if username exists
        # TODO: Read user record
        # Exercise 2
        print("\nTODO: Read user record")
        userKey = ("test", "users", username)
        userRecord = self.client.get(userKey)
        if userRecord:
            record = {}  #Not needed. aerospike will build the record from the key
            #  Get new password
            password = raw_input("Enter new password for " + username + ":")
            # TODO: Register UDF
            # Exercise 2
            print("\nTODO: Register UDF")
            self.client.udf_put("./udf/updateUserPwd.lua")
            # TODO: Execute UDF
            # Exercise 2
            print("\nTODO: Execute UDF");
            new_pass = self.client.apply(userKey, "updateUserPwd", "updatePassword", [password])
            # TODO: Output updated password to the console
            # Exercise 2
            print("\nTODO:  Outout updated password to the console")
            print("Updated Password is: " + new_pass)
        else:
            print("ERROR: User record not found!")
    else:
        print("ERROR: User record not found!")
```

Run the python application and update the password using Record UDF to "passUDF". Using aql, validate that we successfully registered the Lua module and updated the password.

```
aql> show modules
+----------------------------------------------+--------------------+--------+
| hash                                         | module             | type   |
+----------------------------------------------+--------------------+--------+
| "a85ee27925260b0e73d9e1f9315bd0762e98e4ff"   | "updateUserPwd.lua"| "lua"  |
+----------------------------------------------+--------------------+--------+
1 row in set (0.000 secs)

aql> select * from test.users
+----------+--------+--------+-----------+-----------------+------------+---------------+
| username | gender | region | password  | interests       | tweetcount | lasttweeted   |
+----------+--------+--------+-----------+-----------------+------------+---------------+
| "pg"     | "m"    | "s"    | "passUDF" | LIST('["ss"]')  | 1          | 1472692327354 |
+----------+--------+--------+-----------+-----------------+------------+---------------+
1 row in set (0.025 secs)
```

The console should also print the updated password.

In this module, we learned how to
- Code a Record UDF
- Register the UDF module
- Invoke a record UDF to operate on an individual record at the database level.
- **Note:** Record UDFs use *apply* method of Client class, Stream UDFs will use *apply* method of the Query class.

- **Tip:** Aerospike Lua module defines its own record type, map type etc. It uses the userdata type of Lua to make types that can easily move back and forth between client space and UDF space. Methods of these types in the Lua space should not be confused with these types and their methods in the client space. Do not use Lua table type, instead use aerospike map type in your stream UDF functions for aggregation and reduction.

Lab: Queries

At the end of this lab you will be able to

- Create a Secondary Index
- Prepare a Query
- Execute a Query
- Process the result

In this lab, we will augment our "tweetaspike" application to:

- Query Tweets for a given username (equals query)
- Query users based on a number of Tweets (range query)

The lab exercise is at: ~/exercises/Queries/Python

- We will need to create Secondary Indexes for this exercise
- You can create SIs via AQL or in your python client code.
- We will show how to create, display and drop Secondary Indexes via AQL
- We will use the client code to create the actual SIs.

Lab: Exercise 1 - Creating NUMERIC SIs using AQL

We want to create
- a NUMERIC Seconday Index on 'tweetcount' in 'users' set
- AQL Command to create the above SI

  - aql>CREATE INDEX idx_tweets ON test.users (tweetcount) NUMERIC
  - aql>SHOW INDEXES

- To drop (or remove) this index
  - aql> DROP INDEX test.users idx_tweets
  - aql>SHOW INDEX

Lab: Exercise 2 - Creating STRING SIs using AQL

We want to create
- a STRING Seconday Index on 'username' in 'tweets' set
- AQL Command to create the above SI

  - aql>CREATE INDEX idx_user ON test.tweets (username) STRING
  - aql>SHOW INDEXES

- To drop (or remove) this index
  - aql> DROP INDEX test.tweets idx_user
  - aql>SHOW INDEX

- Note: AQL syntax is not case sensitive.

This exercise is to get comfortable with creating, viewing and dropping secondary indexes using AQL. In the next exercise, we will create the secondary indexes from within the Python client application.

Lab: Exercise 3: Python – Query all tweets by a user.

In TweetService.py, locate "Exercise 3" in queryTweetsByUsername()

- Create a STRING Secondary Index on 'username' in the 'tweets' set using the python client function: client.index_string_create().
- Prepare the query by creating a query object on the namespace:set
- Set query filter for username equality using the predicate
  - from aerospike import predicates as p
- Execute the query using:
  - foreach() – process each record via a callback, or
  - results() – get a list of records that match the query criterion
- Execute the query using foreach() in this exercise

```python
def queryTweetsByUsername(self):
    print("\n********** Query Tweets By Username **********\n")
    #  Get username
    username = str()
    username = raw_input("Enter username: ")
    if len(username) > 0:
      try:
        # Create a Secondary Index on tweets
        #Exercise 3
        print("\nTODO: Create a String Secondary Index ")
        #You can also use AQL to create the secondary index.
        #aql> CREATE INDEX idx_user ON test.tweets (username) STRING

        self.client.index_string_create("test", "tweets", "username", "idx_user", None)
        time.sleep(5)  #allow database time to build the index

        #Create the query object
        query = self.client.query('test', 'tweets')
        #Created query to operate on test namespace, tweets set.

        # Exercise 3
        # Set equality Filter on username on the instance of Statement
        print("\nTODO : Set equality Filter on username on the instance of Statement")
        query.where(p.equals('username',username))
        # Execute query
        # Exercise 3
        print("\nTODO : Execute Query")
        # Use the Call back and print Tweets for given Username
        # Exercise 3
        print("\nTODO : Use the Call back and print Tweets for given Username")
        query.foreach(self.printTweet)
      except Exception as e :
        print("error: {0}".format(e), file=sys.stderr)

def printTweet(self, (key, meta, bins)):
    print(bins['tweet'])
```

Lab: Exercise 4: Python – Find users by tweetcount range.

In TweetService.py, locate "Exercise 4" in queryUsersByTweetCount()

- Create a NUMERIC Seconday Index on 'tweetcount' in 'users' set using the python client function: client.index_integer_create().
- Prepare the query by creating a query object on the namespace:set
- Set query filter for tweetcount between num1 and num2 using the predicate
  - from aerospike import predicates as p
- Execute the query using:
  - foreach() – process each record via a callback, or
  - results() – get a list of records that match the query criterion
- Execute the query using foreach() in this exercise

```python
def queryUsersByTweetCount(self):
    print("\n********** Query Users By Tweet Count Range **********\n")
    try:
        # Create a Secondary Index on on tweetcount
        # Exercise 4
        print("\nTODO: Create Integer Secondary Index ")
        #You can also use AQL to create the secondary index
        #aql> CREATE INDEX idx_tweets ON test.users (tweetcount) NUMERIC

        self.client.index_integer_create("test", "users", "tweetcount", "idx_tweets", None)
        time.sleep(5)  #allow database time to build the index

        min = int(raw_input("Enter Min Tweet Count: "))
        max = int(raw_input("Enter Max Tweet Count: "))
        print("\nList of users with " , min , "-" , max , " tweets:\n")
        #Create query
        query = self.client.query("test", "users")
        # Set min--max range Filter on tweetcount
        # Exercise 4
        print("\nTODO: Set min--max range Filter on tweetcount")
        query.where(p.between("tweetcount", min, max))
        # Execute query
        # Exercise 4
        print("\nTODO : Execute Query")
        # Use the Call back and print Tweets for given Username
        # Exercise 4
        print("\nTODO : Iterate through returned RecordSet and for each record, output text in format
ername> has <#> tweets ")
        query.foreach(self.printTweetsByUser)
    except Exception as e :
        print("error: {0}".format(e), file=sys.stderr)

def printTweetsByUser(self, (key,meta,bins)):
    print(bins['username']+" has "+ str(bins['tweetcount'])+ " tweets.")
```

Lab: Aggregations using Stream UDFs

At the end of this lab you will be able to
- Know the situation when a stream UDF is appropriate
- Develop stream User Defined Functions (UDFs) in Lua
- Register a UDF with the cluster
- Invoke a stream UDF in Python
- Correctly handle errors

Prerequisite: Familiarity with Lua syntax – similar to JavaScript – is assumed

In this lab, we will augment our "tweetaspike" application to:
- Aggregate users by region who have a tweetcount between a min and max range.
- Regions are North, South, East, West (n,s,e,w respectively)

The lab exercise is at: ~/exercises/Aggregations/Python
- We will need to create Secondary Indexes for this exercise
- You can create SIs via AQL or in your python client code.
- We will use the client code to create the actual SIs.
- In implementing Stream UDFs, we will have to pay close attention to properly configure Lua system and user paths on client nodes and cluster nodes.

Exercise 1 – Write a Stream UDF

Locate aggregationsByRegion.lua file in the udf subfolder and add lua code to:

```lua
local function aggregate_stats(mapObj,rec)
        -- Exercise 1
        -- NOTE: rec will include 'region' bin of type string and the valid values are 'n', 's', 'e', 'w'
        -- TODO: Examine value of 'region' bin in record rec and increment respective counter in the map
        -- TODO: return updated map
        -- Note: rec is aerospike lua record type. It has various methods available to
        -- get the digest and metadata as well us update the record.
        -- mapObj is what we passed to aggregate_stats as an aerospike lua map type.
end

local function reduce_stats(a,b)
    -- Exercise 1
    -- TODO: Merge values from map b into a
    -- TODO: Return updated map a
end

function sum(stream)
    -- Exercise 1
    -- TODO: Process incoming record stream and pass it to aggregate function, then to reduce function
    --    NOTE: aggregate function aggregate_stats accepts two parameters:
    --      1) A map that contains four variables to store number-of-users counter
    -- for north, south, east and west regions with initial value set to 0

    --      2) function name aggregate_stats -- which will be called for each record as it flows in
    -- TODO: Return reduced value of the map generated by reduce function reduce_stats

        -- You must use the aerospike lua module defined types so you can return back to the client in
        -- recognized types.  Use map type instead of Lua table type here.
end
```

- Code main function 'sum' to process incoming record stream and pass it to aggregate function 'aggregate_stats'.
- Code 'aggregate_stats' to examine value of 'region' bin and increment respective counters.
- Code reduce function 'reduce_stats' to merge maps.

Locate aggregationsByRegion.lua file in the udf subfolder and add lua code to:

- Code main function 'sum' to process incoming record stream and pass it to aggregate function 'aggregate_stats'.
- Code 'aggregate_stats' to examine value of 'region' bin and increment respective counters.
- Code reduce function 'reduce_stats' to merge maps.

```lua
local function aggregate_stats(mapObj,rec)
        -- Exercise 1
        -- NOTE: rec will include 'region' bin of type string and the valid values are 'n', 's', 'e', 'w'
        -- TODO: Examine value of 'region' bin in record rec and increment respective counter in the map
        -- TODO: return updated map
        -- Note: rec is aerospike lua record type. It has various methods available to
        -- get the digest and metadata as well us update the record.

        if rec['region'] == 'e' then
          mapObj['e']= mapObj['e'] + 1
        elseif rec['region'] == 'w' then
          mapObj['w']= mapObj['w'] + 1
        elseif rec['region'] == 'n' then
          mapObj['n']= mapObj['n'] + 1
        elseif rec['region'] == 's' then
          mapObj['s']= mapObj['s'] + 1
        end
        -- mapObj is what we passed to aggregate_stats as an aerospike lua map type.
        return mapObj
end

local function reduce_stats(a,b)
  -- Exercise 1
  -- TODO: Merge values from map b into a
  -- TODO: Return updated map a
  a['e']= a['e'] + b['e']
  a['w']= a['w'] + b['w']
  a['n']= a['n'] + b['n']
  a['s']= a['s'] + b['s']
  --Alternately you can use the merge method of the aerospike map type
  return a
end

function sum(stream)
  -- Exercise 1
  -- TODO: Process incoming record stream and pass it to aggregate function, then to reduce function
  --    NOTE: aggregate function aggregate_stats accepts two parameters:
  --      1) A map that contains four variables to store number-of-users counter
  -- for north, south, east and west regions with initial value set to 0

  --      2) function name aggregate_stats -- which will be called for each record as it flows in
  -- TODO: Return reduced value of the map generated by reduce function reduce_stats

    -- You must use the aerospike lua module defined types so you can return back to the client in
    -- recognized types.  Use map type instead of Lua table type here.
    m = map{e=0,w=0,n=0,s=0}
    return stream : aggregate(m, aggregate_stats) : reduce(reduce_stats)
end
```

Note: In function sum() we use the aerospike Lua type 'map' which is similar to Lua type Table in syntax but is mapped by aerospike between the Lua modules and Aerospike Client. Do not use the Lua Table type to return values to the Client.

Aerospike map type has a merge method that uses a call back for defining the user's merge operation and then iterates over each map entry to merge two maps. User is encouraged to explore various methods of the aerospike Lua types in the documentation.

Exercise 2 – Python: Register & Execute a Stream UDF

In Program.py, ensure that the `config` parameter for opening the client connection to the server

- Identifies system and user lua paths on the client node.
- client user-id on client node has write access to the lua user path

```
config = {'hosts': [('127.0.0.1', 3000)],
            'lua': {'system_path':'/usr/local/aerospike/lua/',
            'user_path':'/usr/local/aerospike/usr-lua/'}}
self.client = aerospike.client(config).connect()
```

In UserService.py, aggregateUsersByTweetCountByRegion():

- Using the udf_put() api, add code to register the UDF.
- Add NUMERIC Secondary Index on tweetcount.
- Prepare query with predicate tweetcount between min and max.
- Apply the stream udf aggregationByRegion module, sum function to the query.
- Examine the results using query.results() and / or query.foreach() API.

In this exercise, use hidden menu options 12 and 23 to create users and tweets data.
Starting at: ~/exercises/Aggregations/Python
$python Program.py
12
..<Enter>
$python Program.py
23
..<Enter>

Locate aggregationsByRegion.lua file in the udf subfolder and add lua code to:

- Code main function 'sum' to process incoming record stream and pass it to aggregate function 'aggregate_stats'.
- Code 'aggregate_stats' to examine value of 'region' bin and increment respective counters.
- Code reduce function 'reduce_stats' to merge maps.

Note on Registering UDFs –Python Client

While Record UDFs run only on the server, stream UDFs run both on server and the client. Both server and client nodes also have Aerospike system Lua modules, server side installed during server installation on server's `/opt/aerospike/sys/udf/lua` directory, client side aerospike Lua modules (not a copy of server side aerospike Lua modules) installed when installing the client at `/usr/local/aerospike/lua` on the client node.

When user registers user UDF in Lua via AQL, it is only added to the server side, user UDF directory `/opt/aerospike/usr/udf/lua`. Depending on the client, for stream UDFs, user must also upload user UDFs to `/usr/local/aerospike/usr-lua` on client node. If using the client api `udf_put()`, ensure client has write privileges on `/usr/local/aerospike/usr-lua`.

You can do: `$sudo chown <user-id> /usr/local/aerospike/usr-lua`

In Python client, when you use `udf_put()` in application code, it will upload the user UDF both on server and client side, provided you have passed the user side configuration when connecting to the client via the `config` parameter. For example, in client application do:

```
config = {'hosts': [('127.0.0.1', 3000)],
          'lua': {'system_path':'/usr/local/aerospike/lua/',
          'user_path':'/usr/local/aerospike/usr-lua/'}}
   self.client = aerospike.client(config).connect()
```

The sample output should be similar to below.

```
Input:7

********** Your Selection: Stream UDF -- Aggregation Based on Tweet Count By Region **********

Enter Min Tweet Count: 3
Enter Max Tweet Count: 6

Aggregating users with  3 - 6  tweets by region:


TODO: Register UDF

TODO: Set min--max range Filter on tweetcount

TODO: Execute aggregate query passing in , .lua filename of the UDF and lua function name.
Total users in East =   337
Total users in West =   277
Total users in North =  329
Total users in South =  299

TODO: Examine returned ResultSet and output result to the console in format "Total Users in <region>: <#>"
[{'s': 299, 'e': 337, 'w': 277, 'n': 329}]
```

One solution is as below.

```python
    def aggregateUsersByTweetCountByRegion(self):
        policy = {}
        lua_file_name = './udf/aggregationByRegion.lua'
        udf_type = 0 # 0 for LUA
        try:
            min = int(raw_input("Enter Min Tweet Count: "))
            max = int(raw_input("Enter Max Tweet Count: "))
            print("\nAggregating users with " , min , "-" , max , " tweets by region:\n")
            # TODO: Register UDF
            # Exercise 2
            # NOTE: UDF registration has been included here for convenience
            # NOTE: The recommended way of registering UDFs in production env is via AQL

            print("\nTODO: Register UDF")

            self.client.udf_put(lua_file_name, udf_type, policy)

            # Create Secondary Index on tweetcount
            # Preferred way is to create once via AQL

            self.client.index_integer_create("test", "users", "tweetcount", "tweetcount_index", None)
            time.sleep(1)  #time to build the index

            # Set min--max range Filter on tweetcount
            # Exercise 2
            print("\nTODO: Set min--max range Filter on tweetcount")

            query = self.client.query("test","users")  #create query object on namespace and set
            query.where(p.between('tweetcount', min, max) )  #we already have SI on tweetcount

            # Execute aggregate query passing in , .lua filename of the UDF and lua function name
            # Exercise 2
            print("\nTODO: Execute aggregate query passing in , .lua filename of the UDF and lua function name.")
            def printResult(result):
              print("Total users in East = ", result['e'])
              print("Total users in West = ", result['w'])
              print("Total users in North = ", result['n'])
              print("Total users in South = ", result['s'])

            query.apply("aggregationByRegion", "sum")

            query.foreach(printResult)

            # Examine returned ResultSet and output result to the console in format \"Total Users in <region>: <#>\"
            # Exercise 2
            print("\nTODO: Examine returned ResultSet and output result to the console in format \"Total Users in
gion>: <#>\"")

            result = query.results()
            print(result)

        except Exception as e :
            print("error: {0}".format(e), file=sys.stderr)
```