AS101 Lab Exercises


Lab: Key-value Operations

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

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:

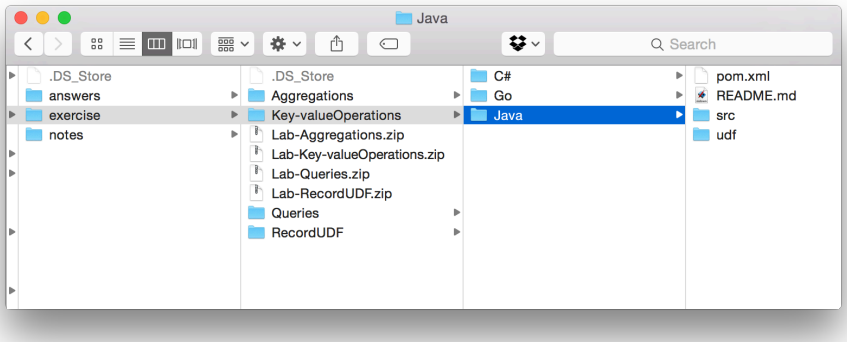
- Answers
- Exercise
- Notes

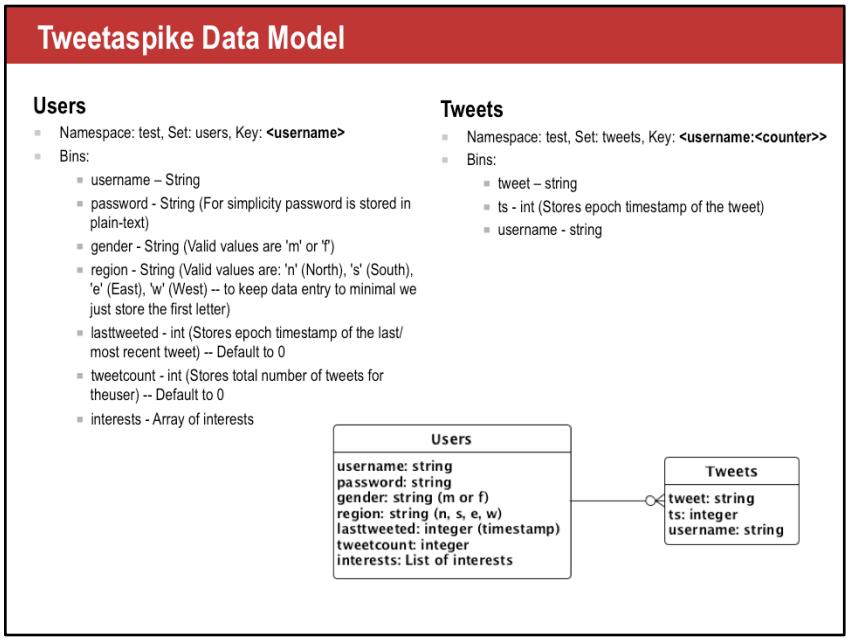
In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Go
- PHP
- Ruby
- Node.js
- Python

The exercises for this module are in the Key-valueOperations directory and your will find a Project/ Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address





Users

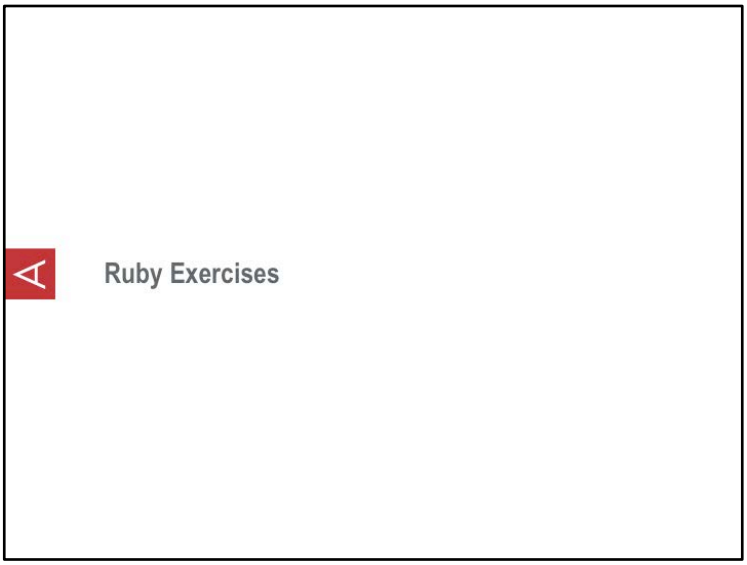
Namespace: test, Set: users, Key: <username>

Bin name	Type	Comment
username	String	
password	String	For simplicity password is stored in plain text
region	String	Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) to keep data entry to minimal we just store the first le] er
las] weeted	Integer	Stores epoch Nmestamp of the last/most recent tweet Default to 0
tweetcount	Integer	Stores total number of tweets for the user – Default 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 Nmestamp of the tweet
username	String	User name of the tweeter


Ruby Exercises

Exercise 1 – Ruby: Connect & Disconnect

Locate the Training module in the Ruby project

1. Create an instance of the Aerospike client in the Training module's init method. Ensure that this connection is created only once
2. Add code to disconnect from the cluster in the Training.finish method. Ensure that this code is executed only once

```
# todo: configure and instantiate the Aerospike client
#@client = # assign the client to the module instance variable
```

```
# todo: close the client's connection to the cluster
```

In this exercise you will connect to a Cluster by creating an Aerospike client instance, using a single IP address and port. These should be to a node in your own cluster.

Ensure that you only create one client instance at the start of the program. The Aerospike is thread safe and creates a pool of worker threads, this means you DO NOT need to create your own connection or thread pool.

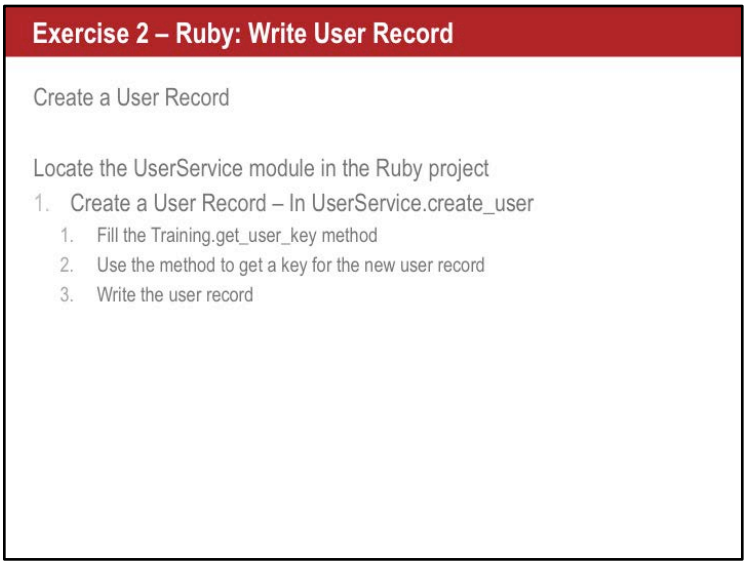
1. In the Training.init, add code similar to this;

```
@client = host ? Client.new(Host.new(host, port)) : Client.new
```

Make sure you have your server up and you know its IP address

2. In Training.finish, add code, similar to this, to disconnect from the cluster. This should only be done once. After close() is called, the client instance cannot be used.

```
@client->close
```



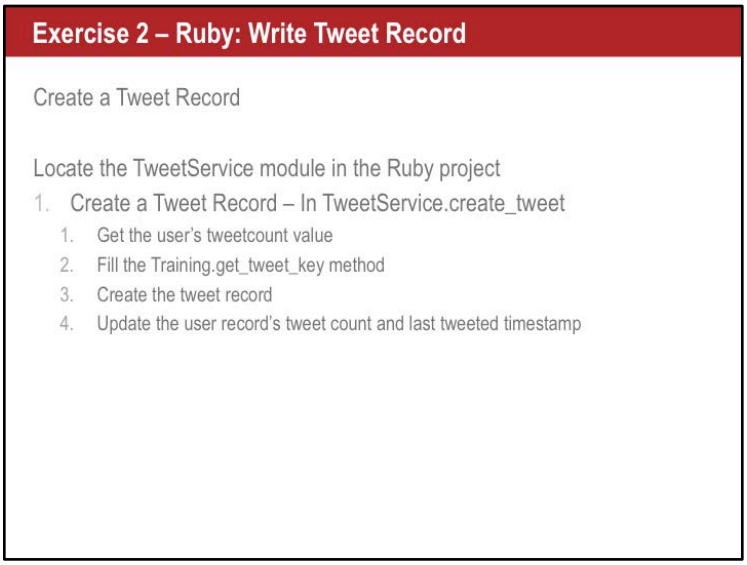
Create a User Record. In UserService.create_user, add code similar to this:

1. Fill the Training.get_user_key method
2. Use the method to get a key for the new user record
3. Write the user record

```
def get_user_key(username)
  Key.new(self.namespace, self.set_name, username)
end
```

```
   def create_user(client)
     puts "\nCreate a new user".colorize(:color => :blue, :mode
=> :bold)
     print 'Enter username (or hit Return to skip):
'.colorize(:blue)
     username = gets.chomp
     bins = { 'username' => username }
     return if username.length < 1
     print "Enter password for #{username}: ".colorize(:blue)
     bins['password'] = gets.chomp
     print "Select gender (f or m) for #{username}:
".colorize(:blue)
     bins['gender'] = gets.chomp
     print "Select region (north, south, east or west) for
#{username}: ".colorize(:blue)
     bins['region'] = gets.chomp
     print "Enter comma-separated interests for #{username}:
".colorize(:blue)
     bins['interests'] = gets.chomp.split(',')

     print "Creating user record >".colorize(:color
=> :black, :mode => :bold)
     begin
       key = Key.new(self.namespace, self.set_name, username)
       client.put(key, bins, self.write_policy)
       self.yep
     rescue
       self.nope
       puts "Connection to Aerospike cluster failed! Please
check the server settings and try again!".colorize(:color
=> :red, :mode => :bold)
     end
   end
```



Create a Tweet Record. In TweetService.create_tweet, add code similar to this:

1. Get the user's tweetcount value
2. Fill the Training.get_tweet_key method
3. Create the tweet record
4. Update the user record's tweet count and last tweeted timestamp

```
  def get_tweet_key(username, id)
    Key.new(self.namespace, 'tweets', "#{username}:#{id}")
  end

    def create_tweet(client, username = nil)
      puts "\nCreate a new tweet".colorize(:color => :blue, :mode
=> :bold)
      unless username
        print "Enter username (or hit Return to skip): ".colorize(:blue)
        username = gets.chomp
      end
      unless username == ''
        key = self.get_user_key(username)
        rec = client.get(key, ['tweetcount'])
        bins = rec.bins unless rec.nil?
        tweet_count = bins.nil? ? 1 : bins['tweetcount'] + 1
      end
      ts = (Time.now.to_f * 1000).round
      bins = {'ts' => ts}
      print "Enter tweet for #{username}: ".colorize(:blue)
      bins['tweet'] = gets.chomp
      bins['username'] = username
      print "Creating tweet record >".colorize(:color => :black, :mode
=> :bold)
      key = self.get_tweet_key(username, tweet_count)
      begin
        client.put(key, bins)
        self.yep
      rescue Exception => e
        self.nope
        puts "Failed to create the tweet".colorize(:color => :red, :mode
=> :bold)
        pp e
        return
      end
      key = self.get_user_key(username)
      bins = {'tweetcount' => tweet_count, 'lasttweeted' => ts}
      print "Updating the user record >".colorize(:color => :black, :mode
=> :bold)
      begin
        client.put(key, bins)
        self.yep
      rescue
        self.nope
        print "Failed to update the user record >".colorize(:color
=> :black, :mode => :bold)
      end
    end
```

Exercise 2 Ruby: Read Records

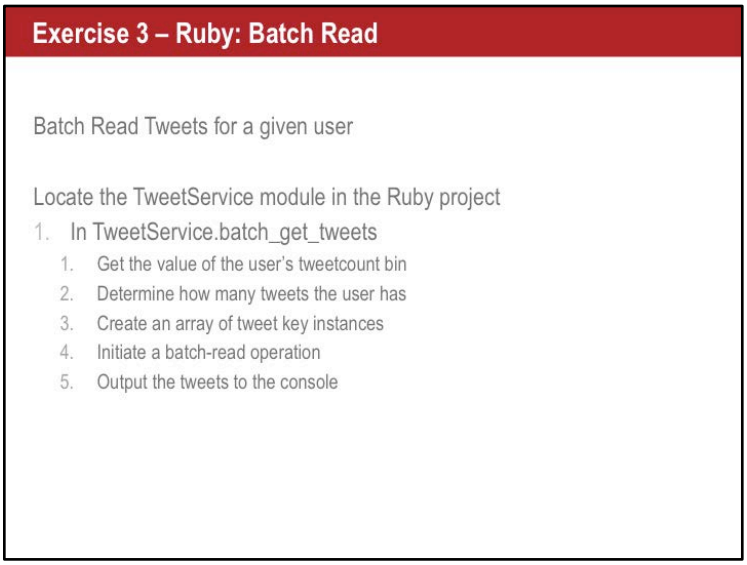
Read User Record

Locate the UserService module in the Ruby project

1. Read a User Record – In UserService.get_user
   1. Read the user record from the cluster

In UserService.get_user, add code, similar to this, to:

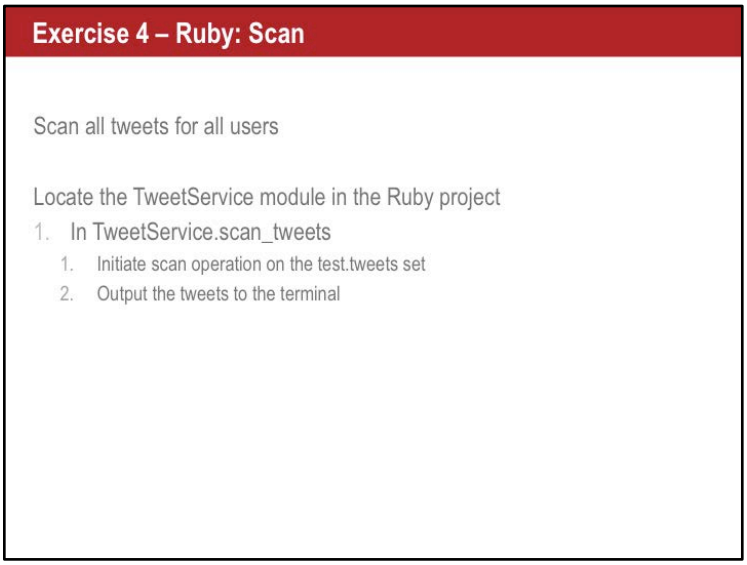
```
  def get_user(client)
      puts "\nCreate a user".colorize(:color
=> :blue, :mode => :bold)       print 'Enter username
(or hit Return to skip): '.colorize(:blue)
username = gets.chomp
      key = self.get_user_key(username)
      rec = client.get(key)
      if rec
        puts "Record.key (Key)".colorize(:mode
=> :bold)
        pp rec.key
        puts "Record.generation
(Fixnum)".colorize(:mode => :bold)         pp
rec.generation
        puts "Record.expiration
(Fixnum)".colorize(:mode => :bold)         pp
rec.expiration
        puts "Record.bins (Hash)".colorize(:mode
=> :bold)
        pp rec.bins
      else
        puts "There is no such user
#{username}".colorize(:red)       end
    end
```



Read all the tweets for a given user. In TweetService.batch_get_tweets, add code similar to this:

1. Get the value of the user's tweetcount bin
2. Determine how many tweets the user has
3. Create an array of tweet key instances
4. Initiate a batch-read operation
5. Output the tweets to the console

```
    def batch_get_tweets(client, username = nil)
      puts "\nGet the user's tweet".colorize(:color => :blue, :mode
=> :bold)
      unless username
        print "Enter username (or hit Return to skip): ".colorize(:blue)
        username = gets.chomp
      end
      unless username == ''
        key = self.get_user_key(username)
        rec = client.get(key, ['tweetcount'])
        unless rec
          puts "There is no such user #{username}".colorize(:red)
          return
        end
        bins = rec.bins
        tweet_count = bins.nil? ? 0 : bins['tweetcount']
        keys = []
        (1..tweet_count).each do |i|
          keys.push(self.get_tweet_key(username, i))
        end
        print "Batch-reading the user's tweets >".colorize(:color
=> :black, :mode => :bold)
        begin
          recs = client.batch_get(keys)
          self.yep
        rescue Exception => e
          self.nope
          puts "Failed to batch-read the tweets for
#{username}".colorize(:color => :red, :mode => :bold)
          pp e
          return
        end
        puts "Here are #{username}'s tweets:".colorize(:color
=> :blue, :mode => :bold)
        recs.each do |r|
          puts r.bins['tweet']
        end
      end
    end
```



Scan all the tweets for all users – warning – there could be a large result set.

In the TweetService.scan_tweets, add code similar to this:

1. Initiate scan operation on the test.tweets set
2. Output the tweets to the terminal

```
    def scan_tweets(client)
      puts "\nScan for tweets".colorize(:color
=> :blue, :mode => :bold)
      begin
        policy = ScanPolicy.new(:fail_on_cluster_change =>
true)
        recordset = client.scan_all(self.namespace,
'tweets', [], policy)

        recordset.each do |rec|
          puts rec.bins['tweet']
        end
      rescue
        puts "Failed to scan test.tweets".colorize(:color
=> :red, :mode => :bold)
      end
    end
```

Exercise 5 – Ruby: Read-modify-write

Update the User record with a new password ONLY if the user record is unmodified.

Locate the UserService module in the Ruby project

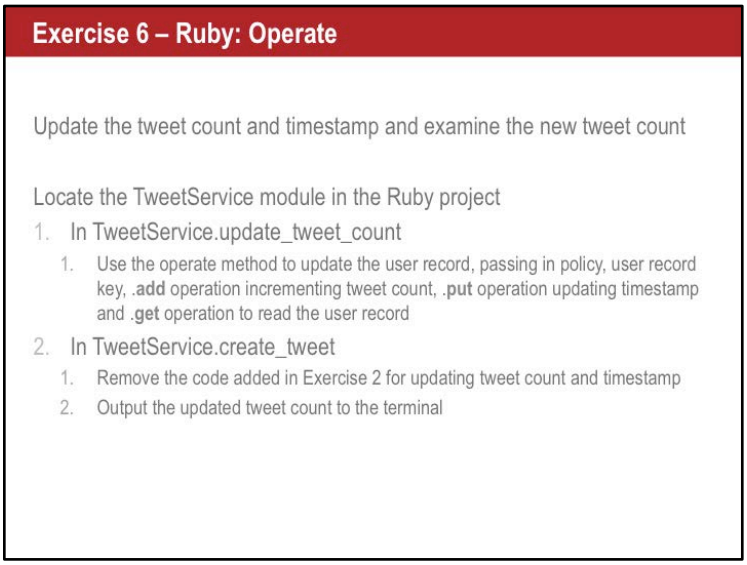
1. In UserService.check_and_set_password
    1. Get the generation of the user record
    2. Instantiate a WritePolicy
    3. Set the WritePolicy.generation to the value read from the user record
    4. Set WritePolicy's generation_policy to GenerationPolicy::EXPECT_GEN_EQUAL
    5. Update the user record with the new password

Update the User record with a new password ONLY if the User record is une modified In UserService.check_and_set_password, add code similar to this:

1. Get the generation of the user record
2. Instantiate a WritePolicy
3. Set the WritePolicy.generation to the value read from the user record
4. Set WritePolicy's generation_policy to GenerationPolicy::EXPECT_GEN_EQUAL
5. Update the user record with the new password

```
        key = self.get_user_key(username)
        rec = client.get_header(key)
        generation = rec.generation


        write_policy = WritePolicy.new
        write_policy.generation_policy =
GenerationPolicy::EXPECT_GEN_EQUAL
        write_policy.generation = generation
        client.put(key, bins, write_policy)
```



In TweetService.update_tweet_count

1. Use the operate method to update the user record, passing in policy, user record key, .add operation incrementing tweet count, .put operation updating timestamp and .get operation to read the user record

```
ops = [
 Operation.add(Bin.new('tweetcount', 1)),
 Operation.put(Bin.new('lasttweeted', (Time.now.to_f *
 1000).round))
]
client.operate(key, ops)
```

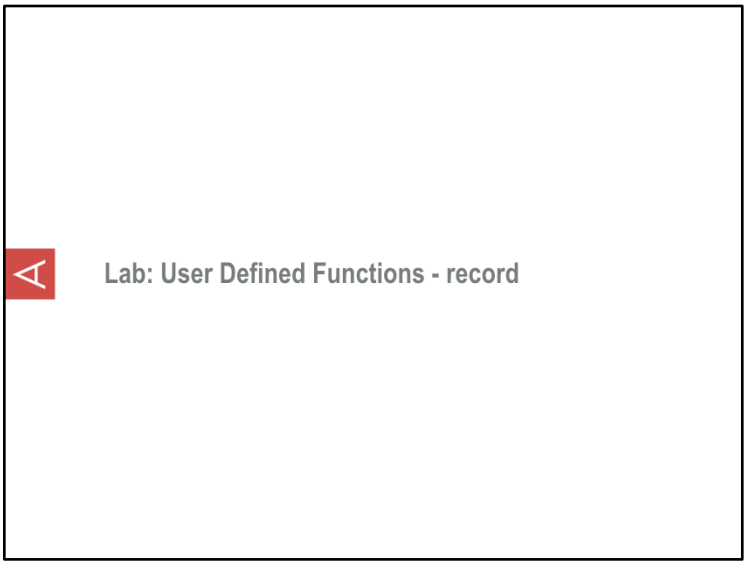
In TweetService.create_tweet

1. Remove the code added in Exercise 2 for updating tweet count and timestamp
2. Output the updated tweet count to the terminal

Summary

You have learned how to:

- Connect to Cluster
- Write and Read Records
- Batch Read Records
- Read-Modify-Write
- Operate
- Handle errors correctly


Lab: User Defined Functions - record

Objective

After successful completion of this Lab module you will have:

- Coded a Record UDF
- Registered the UDF with a cluster
- Invoked the UDF from your C#, Go, PHP, Ruby, Node.js or Java application

Lab Overview

The lab exercise augments "tweetaspike" by using a Record UDF. Here we will focus on a Record UDF that updates user password.

You will:

- Write a user defined function, in Lua, to update the user password
- Register the UDF
- Execute the UDF from your application

The application shell is located in your cloned GitHub directory

```
~/exercises/RecordUDF/<language>
```

Make sure you have your server up and you know its **IP address**

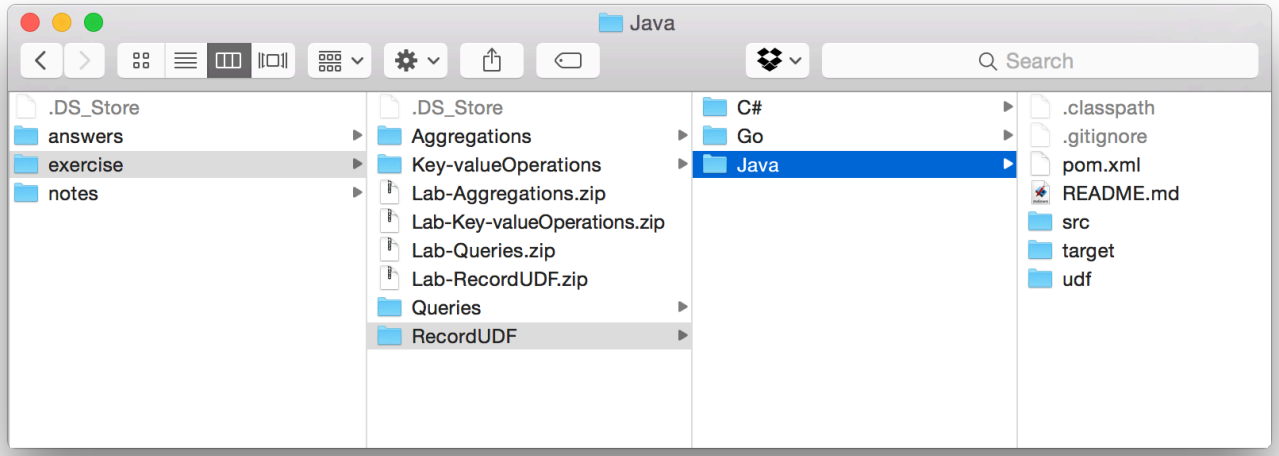
In your cloned or downloaded repository you will find the following directories:

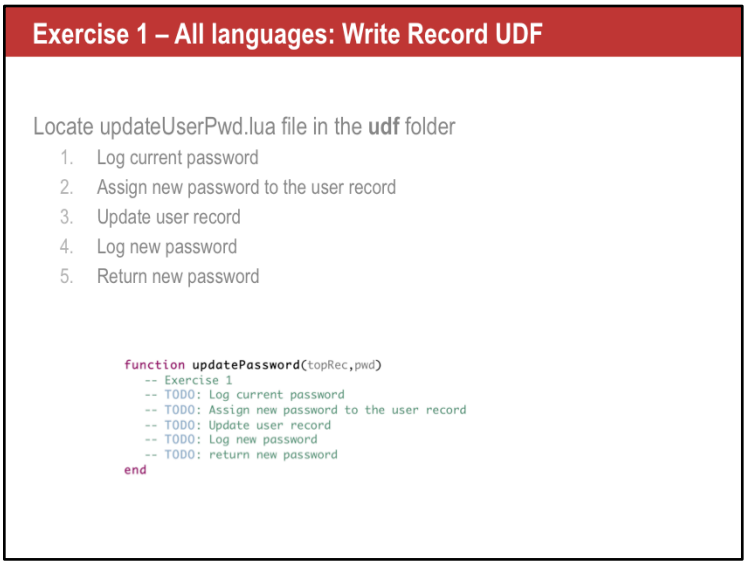
- Answers
- Exercise
- Notes

In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Node.js
- Go
- PHP
- Ruby

The exercises for this module are in the UDF directory and your will find a Project/SoluNon/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

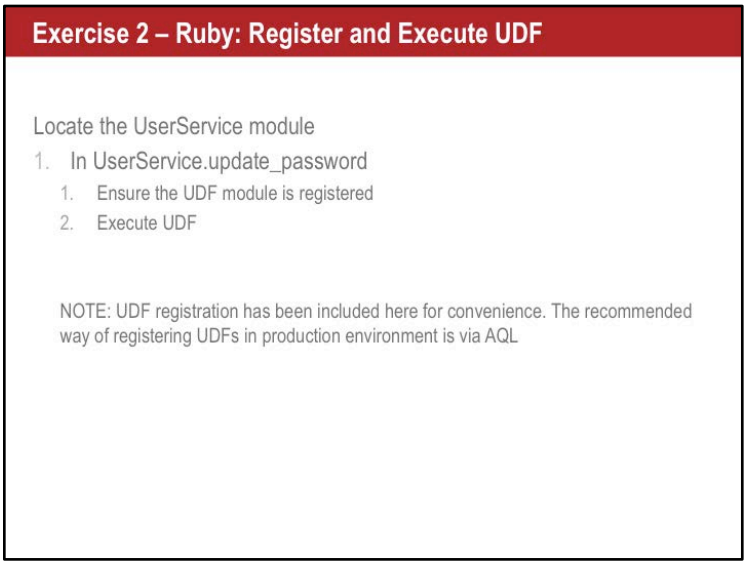




In this exercise you will create a record UDF that:

1. Logs the current password
2. Assigns a new password to the record, passed in via the pwd parameter
3. Updates the user record by calling **aerospike:update**(topRec)
4. Logs the new password
5. Returns the new password to the client

```
function updatePassword(topRec,pwd)
   -- Log current password
   debug("current password: " .. topRec['password'])
   -- Assign new password to the user record
   topRec['password'] = pwd
   -- Update user record
   aerospike:update(topRec)
   -- Log new password
   debug("new password: " .. topRec['password'])
   -- return new password
   return topRec['password']
end
```



In this exercise you will register and invoke the UDF created in Exercise 1.

We will programma cally register the UDF for convenience.

In UserService.update_password, locate these comments and add your code:

1. Ensure the UDF module is registered

```
task = client.register_udf_from_file(module_path,
module_name, Language::LUA)
task.wait_till_completed
```

2. Execute the UDF passing the new password as a parameter, to the UDF

```
client.execute_udf(key, "updateUserPwd", "updatePassword",  [new_password])
```

Summary

You have learned:

- Code a record UDF
- Register the UDF module
- Invoke a record UDF


Lab: Queries

Objectives

After successful completion of this Lab module you will have:

- Created a secondary index
- Prepared a statement
- Executed a query
- Processed the results

Lab Overview

The Lab exercises augment the"tweetaspike" application by allowing us to:

1) Query Tweets for a given username
2) Query users based on number of Tweets

The application shell is located in your cloned GitHub directory

`~/exercises/Queries/<language>`

Make sure you have your server up and you know its **IP address**

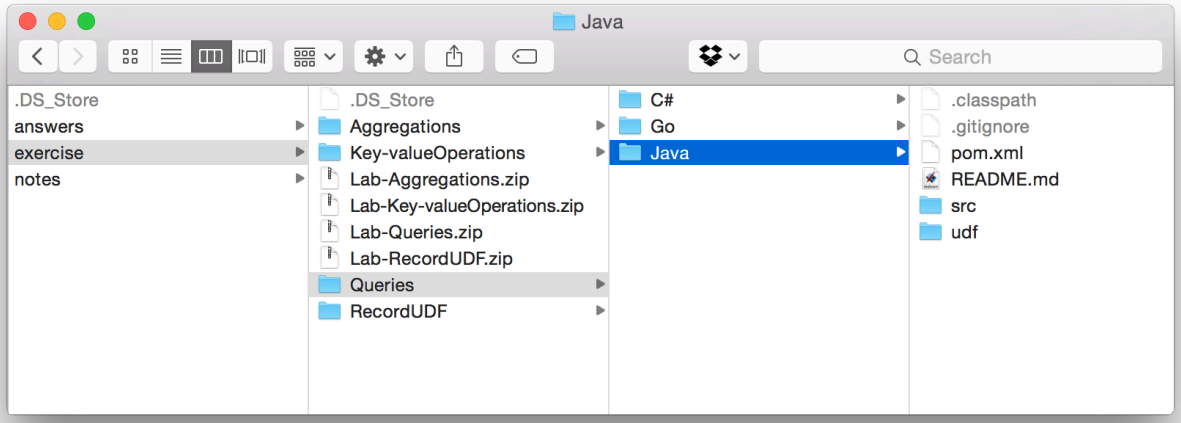
On your cloned or downloaded repository, you will find the following directories:

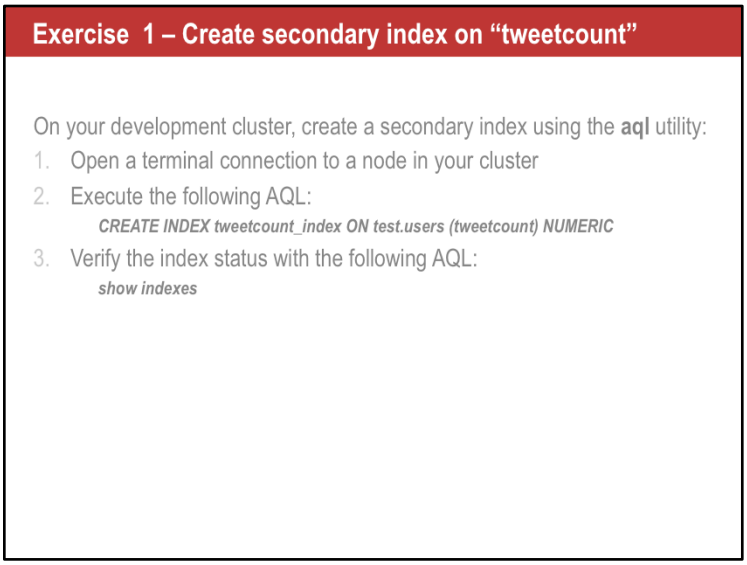
- Answers
- Exercise
- Notes

In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Go
- Node.js
- PHP
- Python

The exercises for this module are in the Queries directory and your will find a Project/SoluNon/ Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.



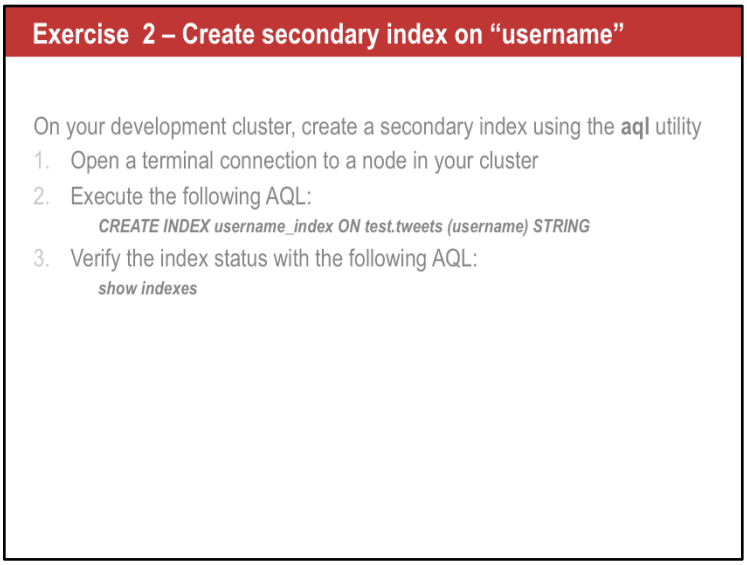


Logon on to your server instance and run **aql** to create a numeric index on *tweetcount*.
At the prompt, enter the command:

CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC

Verify that the index has been created with the command:

show indexes

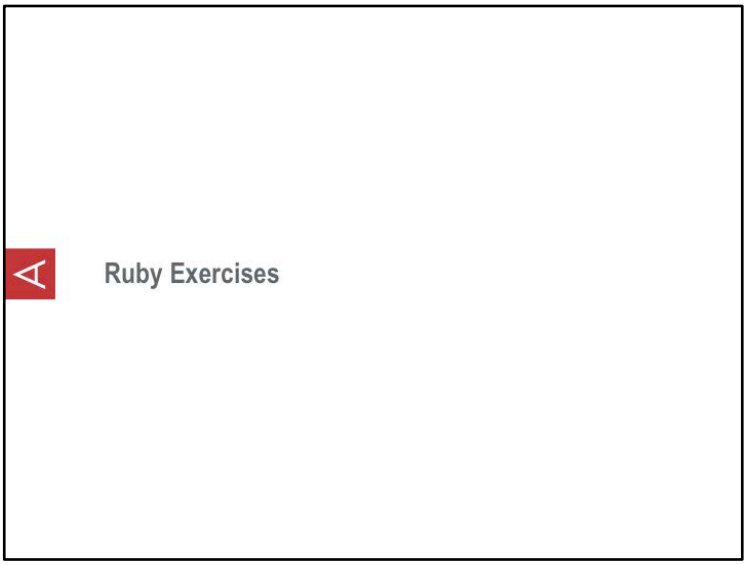


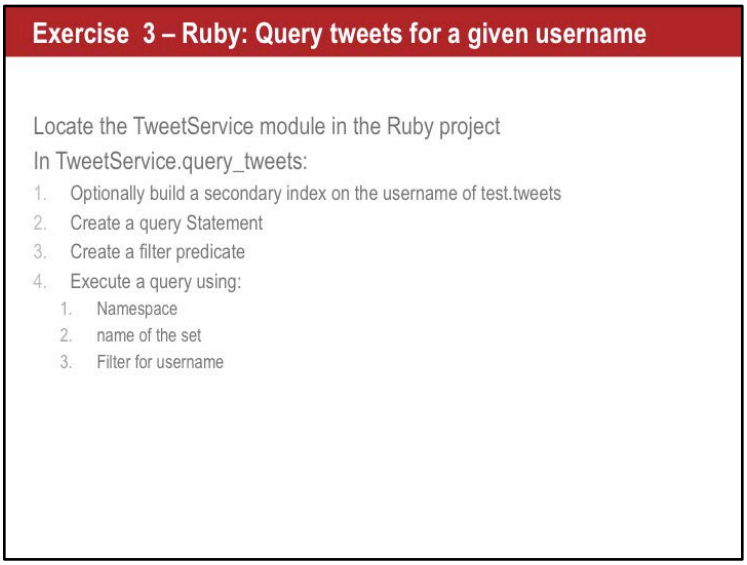
Logon on to your server instance and run **aql** to create a string index on *username*. At the prompt, enter the command:

CREATE INDEX username_index ON test.tweets (username) STRING

Verify that the index has been created with the command:

show indexes

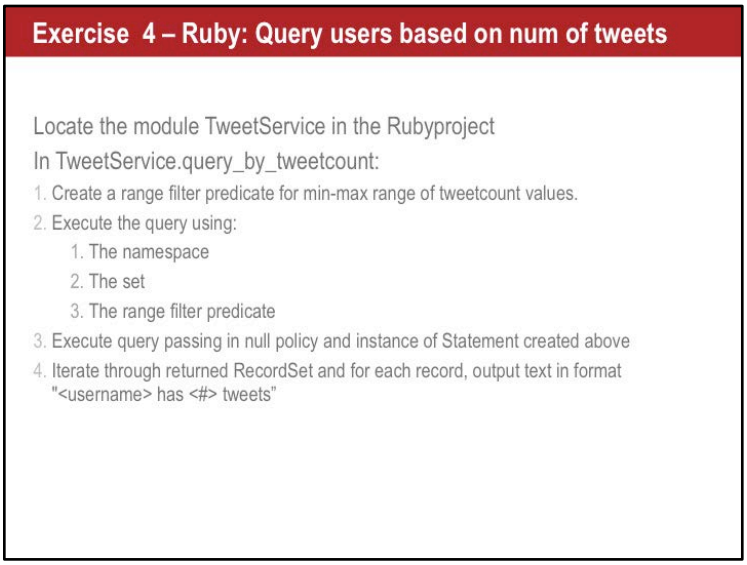

Ruby Exercises



In TweetService.query_tweets addyour code:

1. Create a Filter predicate
2. Execute a query using:
    1. the Namespace
    2. the Set name
    3. the Filter predicate to qualify the user name

```
statement = Statement.new('test', 'tweets', ['tweet'])
statement.filters << Filter.Equal('username', username)
results = client.query(statement)
```



In TweetService.query_by_tweet_count, add your code:

1. Create a range filter predicate for min-max tweetcount.
2. Execute the query using:
    1. The namespace
    2. The set
    3. The range filter predicate
3. Execute query passing in null policy and instance of Statement created above
4. Iterate through returned RecordSet and for each record, output text in format "<username> has <#> tweets"

```
statement = Statement.new('test', 'users', ['username', 'tweetcount'])
statement.filters << Filter.Range('tweetcount', min, max) unless min == 0
&& max == 0
results = client.query(statement)
```

Summary

You have learned:

- How to create a secondary index
- How to create a Statement
- Execute a query on a secondary index
- Process the results from a query


Lab: Aggregations

Objective

After successful completion of this Lab module you will have:

- Coded a Stream UDF
- Register the UDF with a cluster
- Executed Aggregation from your C#, PHP, Node.js or Java application

Lab Overview

The lab exercise augments “tweetaspike” by using a Stream UDF.
Here we will create a Stream UDF that aggregates number of users with tweet count between min-max range by region – North, South, East and West .

The application shell is located in your cloned GitHub directory

```
~/exercises/Aggregations/<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:

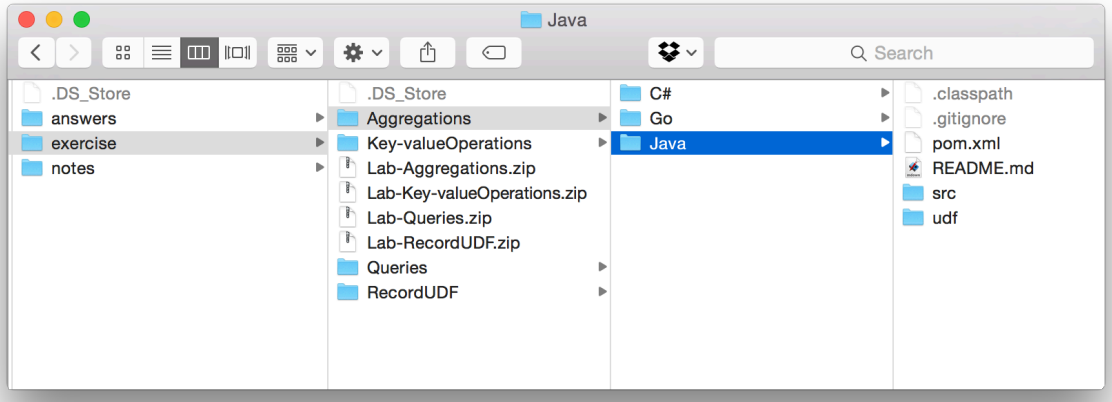
- Answers
- Exercise
- Notes

In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Node.js
- PHP
- Python

The exercises for this module are in the Aggrega ons directory and your will find a Project/SoluNon/ Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address



Exercise 1 – Write Stream UDF

Locate aggregationByRegion.lua file under udf folder in AerospikeTraining Solution

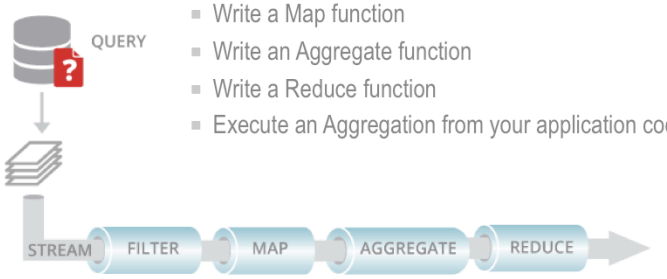
1. Code main function 'sum' to process incoming record stream and pass it to aggregate function 'aggregate_stats', then to reduce function 'reduce_stats'
2. Code aggregate function 'aggregate_stats' to examine value of 'region' bin and increment respective counters
3. Code reduce function 'reduce_stats'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

In this exercise you will create a Stream UDF module that:

- Aggregates (sums) tweets by region – The aggregate_stats() funcNon is invoked one for each element in the stream.
- Reduces the aggrega ons into a single Map of values – The reduce_stats() funcNon is invoked once for each data parNNon, once for each node in the cluster, and finally once on the client.
- The sum() funcNon configures the stream processing, and it is the funcNon invoked by the Client.

```
local function aggregate_stats(map,rec)
  -- Examine value of 'region' bin in record rec and increment respective counter in the map
  if rec.region == 'n' then
      map['n'] = map['n'] + 1
  elseif rec.region == 's' then
      map['s'] = map['s'] + 1
  elseif rec.region == 'e' then
      map['e'] = map['e'] + 1
  elseif rec.region == 'w' then
      map['w'] = map['w'] + 1
  end
  -- return updated map
  return map
end
local function reduce_stats(a,b)
  -- Merge values from map b into a
  a.n = a.n + b.n
  a.s = a.s + b.s
  a.e = a.e + b.e
  a.w = a.w + b.w
  -- Return updated map a
  return a
end
function sum(stream)
  -- Process incoming record stream and pass it to aggregate function, then to reduce function
return stream : aggregate(map{n=0,s=0,e=0,w=0},aggregate_stats) : reduce(reduce_stats)
end
```

Summary



You have learned how to:

- Write a Stream UDF
- Write a Filter function
- Write a Map function
- Write an Aggregate function
- Write a Reduce function
- Execute an Aggregation from your application code


AEROSPIKE