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:

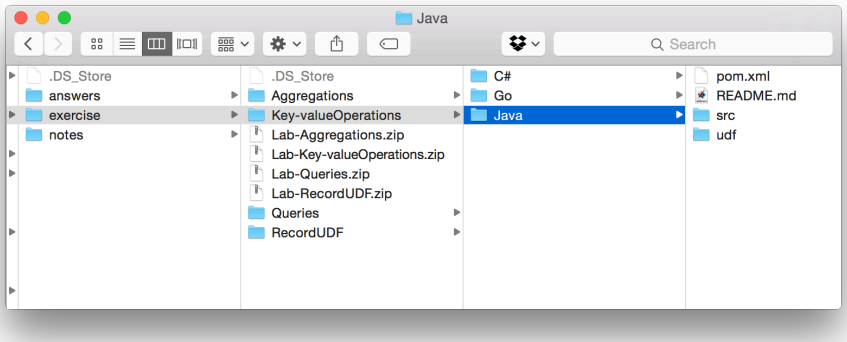
- Answers
- Exercise
- Notes

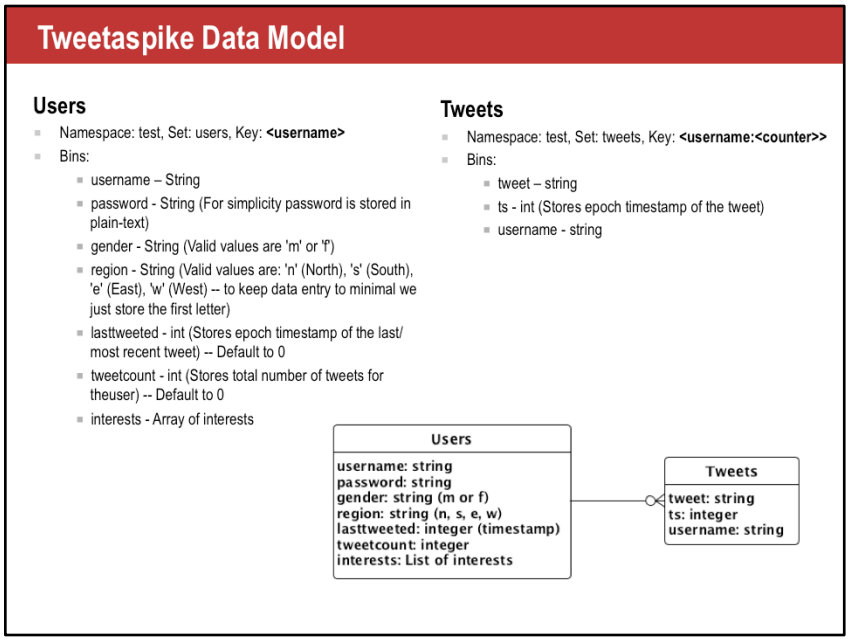
In the exercise directory, select the subdirectory for your programming language:

- 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

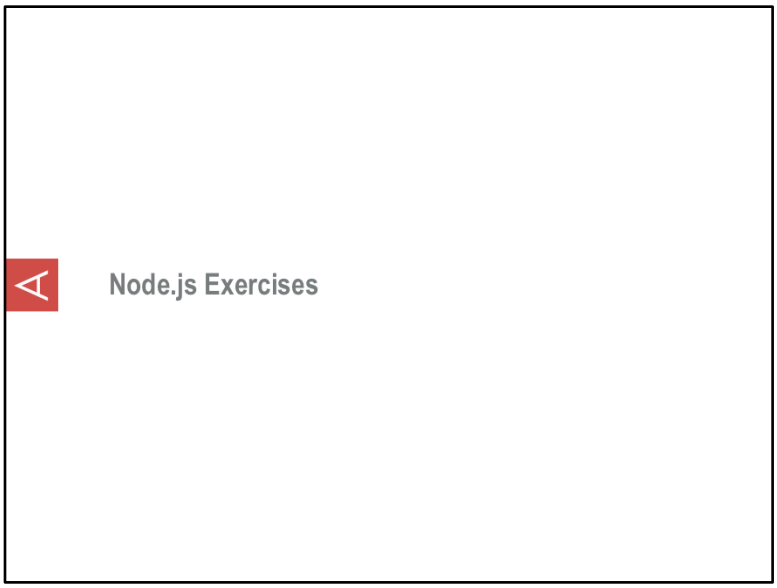
Namespace: test, Set: users, Key: <username>

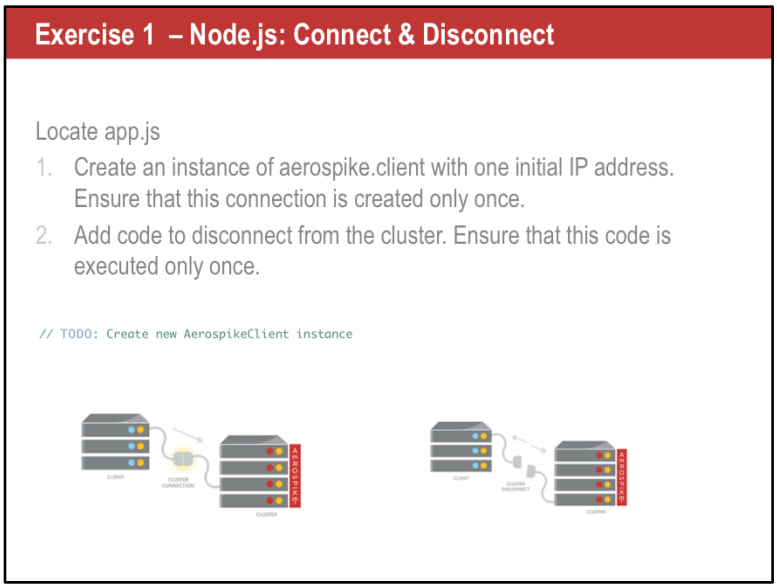
Bin name	Type	Comment
username	String	
password	String	For simplicity password is stored in plain text
region	String	Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) to keep data entry to minimal we just store the first le] er
las] weeted	Integer	Stores epoch Nmestamp of the last/most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


Node.js Exercises



In this exercise you will connect to a Cluster by crea ng an Aerospike client instance, passing a single IP address and port to the constructor. The IP address and port 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 client is thread safe and creates a pool of worker threads, this means you DO NOT need to create your own connecNon or thread pool.

1. In app.js add code similar to this;

```
// Connect to the Aerospike Cluster
var client = aerospike.client({
    hosts: [ { addr: '172.16.159.172', port: 3000 } ]
}).connect(function(response) {
    // Check for errors
    if ( response.code == aerospike.status.AEROSPIKE_OK ) {
      // Connection succeeded
      console.log("Connection to the Aerospike cluster succeeded!");
    }
    else {
      // Connection failed
      console.log("Connection to the Aerospike cluster failed. Please check cluster IP and Port settings and try again.");
      process.exit(0);
    }
});
```

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

2. Add a *process.on* exit funcNon and call close() to disconnect from the cluster. This should only be done once. Ae er close() is called, the client instance cannot be used.

```
// Setup tear down
process.on('exit', function() {
  if (client != null) {
    client.close();
    // console.log("Connection to Aerospike cluster closed!");
  }
});
```

Exercise 2 – Node.js: Write Records

Create a User Record and Tweet Record

Locate user_service.js

1. Create a User Record – In exports.createuser
   1. Create Key and Bin instances for the User Record
   2. Write User Record

Create a User Record. In exports.createUser, add code similar to this:

1. Create Key and Bin instances

```
var key = {
  ns:  "test",
  set: "users",
  key: answers.username
};

var bins = {
  username: answers.username,
  password: answers.password,
  gender: answers.gender,
  region: answers.region,
  lasttweeted: 0,
  tweetcount: 0,
  interests: answers.interests.split(",")
};
```

2. Write a user record using the Key and Bins

```
client.put(key, bins, function(err, rec, meta) {
  // Check for errors
  if ( err.code === 0 ) {
    console.log("INFO: User record created!");

    // Create tweet record
    tweet_service.createTweet(client);
  }
  else {
    console.log("ERROR: User record not created!");
    console.log(err);
  }
});
```

Exercise 2 – ..Cont Node.js: Write Records..

Create a User Record and Tweet Record

Locate tweet_service.js

1. Create a Tweet Record – In export.createTweet
   1. Create Key and Bin instances for the Tweet Record
   2. Write Tweet Record
   3. Update tweet count and last tweeted timestamp in the User Record

Create a Tweet Record. In exports.createTweet add code similar to this:

1. Create Key and Bin instances

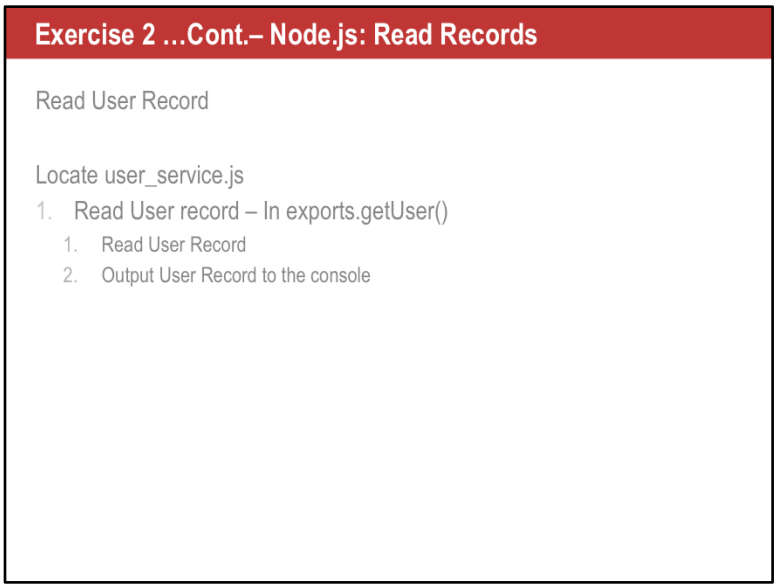
```
// Write Tweet record
var tweet_key = {
  ns:  "test",
  set: "tweets",
  key: userrecord.username + ":" + tweet_count
};

var bins = {
  username: userrecord.username,
  tweet: answer.tweet,
  ts: ts
};
```

2. Write a tweet record using the Key and Bins

```
client.put(tweet_key, bins, function(err, tweetrecord, meta) {
  // Check for errors
  if ( err.code === 0 ) {
    console.log("INFO: Tweet record created!");
```

3. Update the user record with tweet count

```
    // Update tweetcount and last tweet'd timestamp in the user record
    updateUser(client, user_key, ts, tweet_count);
  }
  else {
    console.log("ERROR: Tweet record not created!");
    console.log("",err);
  }
```



Read a User Record. In exports.getUser, add code, similar to this, to:

1. Read a User record
2. Output the User record to the console

```
// Read User record
var key = {
  ns:  "test",
  set: "users",
  key: answer.username
};

client.get(key, function(err, rec, meta) {
  // Check for errors
  if ( err.code === 0 ) {
    console.log("INFO: User record read successfully! Here are the details:");
    console.log("username:   " + rec.username);
    console.log("password:   " + rec.password);
    console.log("gender:     " + rec.gender);
    console.log("region:     " + rec.region);
    console.log("tweetcount: " + rec.tweetcount);
    console.log("lasttweeted: " + rec.lasttweeted);
    console.log("interests:  " + rec.interests);
  }
  else {
    console.log("ERROR: User record not found!");
  }
});
```

Exercise 3 – Node.js: Batch Read

Batch Read tweets for a given user

Locate user_service.js

1. In exports.batchGetUserTweets
   1. Read User Record
   2. Determine how many tweets the user has
   3. Create an array of tweet Key instances -- keys[tweetCount]
   4. Initiate Batch Read operation
   5. Output tweets to the console

Read all the tweets for a given user. In the funcNon exports.batchGetUserTweets, add code similar to this:

1. Read a user record

```
// Read User record
var key = {
  ns:  "test",
  set: "users",
  key: answer.username
};

client.get(key, function(err, userrecord, meta) {
  // Check for errors
  if ( err.code === 0 ) {
```

2. Get the tweet count

```
    var tweet_count = userrecord.tweetcount;
    var tweet_keys = [];
```

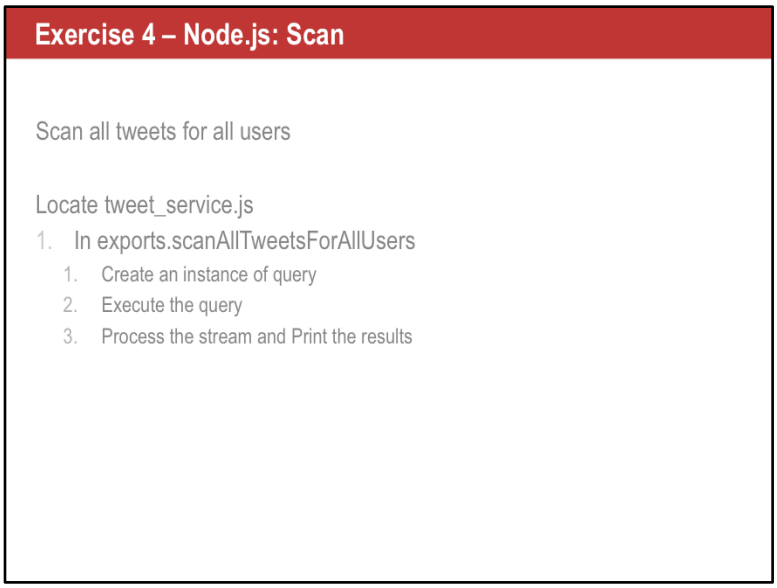
3. Create a “list” of tweet keys

```
    for(var i=1;i<=tweet_count;i++)  {
      tweet_keys.push({ns: "test", set: "tweets", key: answer.username + "
    }
```

4. Perform a Batch opera on to read all the tweets

5. Then print out the tweets

```
    client.batchGet(tweet_keys, function (err, results) {
      // Check for errors
      if ( err.code === 0 ) {
        for(var j=0;j<results.length;j++)  {
          console.log(results[j].record.tweet);
        }
      }
      else {
        console.log("ERROR: Batch Read Tweets For User failed\n", err);
      }
    });

  }
  else {
    console.log("ERROR: User record not found!");
  }
});

  });
```



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

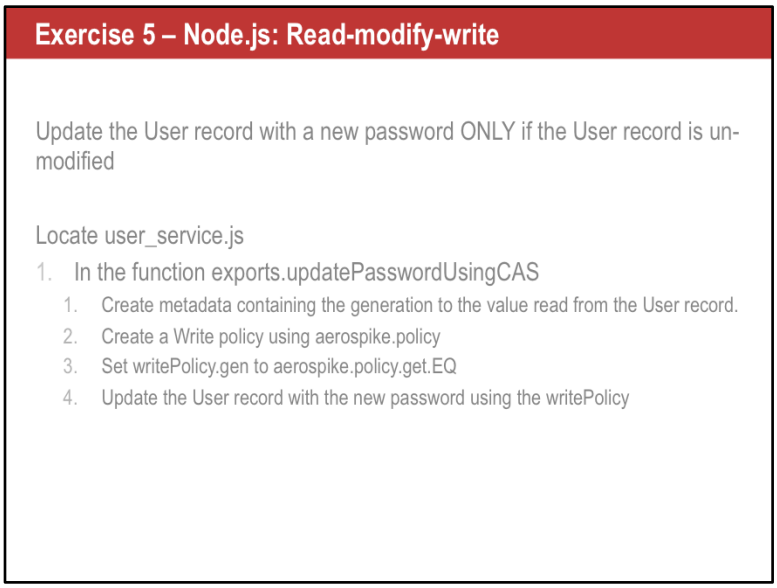
In the funcNon exports.scanAllTweetsForAllUsers, add code similar to this:

1. Create a instance of a query
2. Execute the query

```
var query = client.query('test', 'tweets');
var stream = query.execute();
```

3. Process the stream and print the results

```
stream.on('data', function(record){
     console.log(record.bins.tweet);
});
stream.on('error', function(err){
     console.log('ERROR: Scan All Tweets For All Users failed: ',err);
});
stream.on('end', function(){
     // console.log('INFO: Scan All Tweets For All Users completed!');
});
```

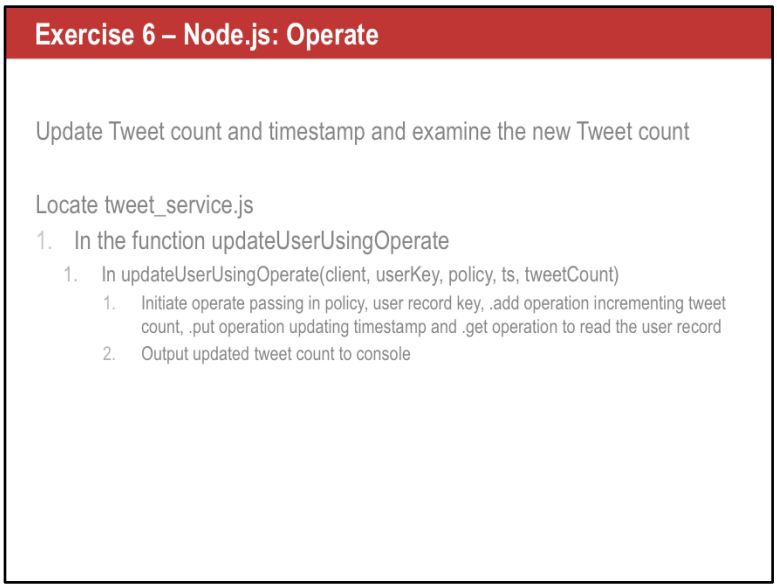


Update the User record with a new password ONLY if the User record is un modified

In exports.updatePasswordUsingCAS, add code similar to this:

1. Create metadata containing the genera on to the value read from the User record.
2. Create a Write policy using aerospike.policy
3. Set writePolicy.gen to aerospike.policy.get.EQ
4. Update the User record with the new password using the writePolicy

```
// Set the generation count to the current one from the user record.
// Then, setting writePolicy.gen to aerospike.policy.gen.EQ will ensure we don't have a 'dirty-
read' when updating user's password
var metadata = {
  gen: meta.gen
}

var writePolicy = aerospike.policy;
writePolicy.key = aerospike.policy.key.SEND;
writePolicy.retry = aerospike.policy.retry.NONE;
writePolicy.exists = aerospike.policy.exists.IGNORE;
writePolicy.commitLevel = aerospike.policy.commitLevel.ALL;
// Setting writePolicy.gen to aerospike.policy.gen.EQ will ensure we don't have a 'dirty-read'
when updating user's password
writePolicy.gen = aerospike.policy.gen.EQ;

var bin = {
  password: answer2.password
};

client.put(key, bin, metadata, writePolicy, function(err, rec) {
  // Check for errors
  if ( err.code === 0 ) {
    console.log("INFO: User password updated successfully!");
  }
  else {
    console.log("ERROR: User password update failed:\n", err);
  }
});
```



Aerospike can perform mulNple opera ons on a record in one transacNon. Update the tweet count and Nmestamp in a user record and read the new tweet count.

In updateUser

1. Comment out the code added in exercise 2

2. Uncomment the line:

```
// TODO: Update tweet count and last tweeted timestamp in the user record using operate
// Exercise 6
// console.log("TODO: Update tweet count and last tweeted timestamp in the user record using
operate");
// updateUserUsingOperate(client, user_key, ts);
```

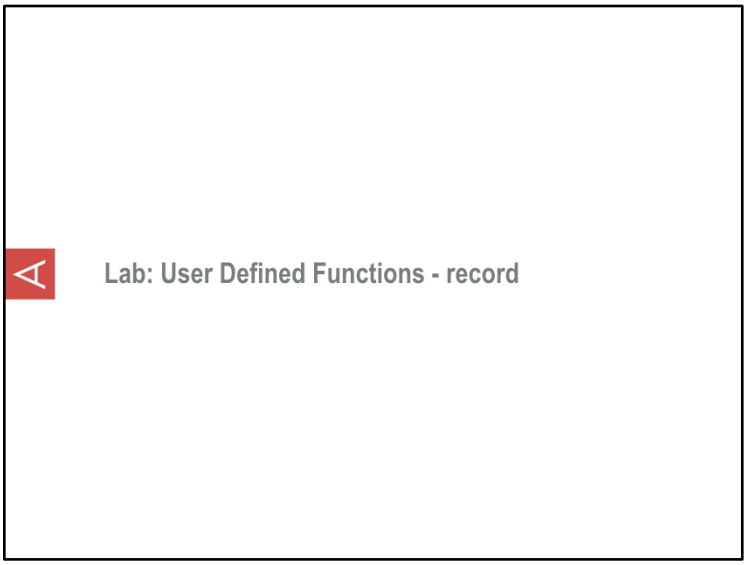
3. In updateUserUsingOperate , add code similar to this:

```
// Update User record
var operator = aerospike.operator;
var operations = [operator.incr('tweetcount', 1),operator.write('lasttweeted',
ts),operator.read('tweetcount')];
client.operate(user_key, operations, function(err, bins, metadata, key) {
  // Check for errors
  if ( err.code === 0 ) {
    console.log("INFO: The tweet count now is: " + bins.tweetcount);
  }
  else {
    console.log("ERROR: User record not updated!");
    console.log(err);
  }
});
```

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**

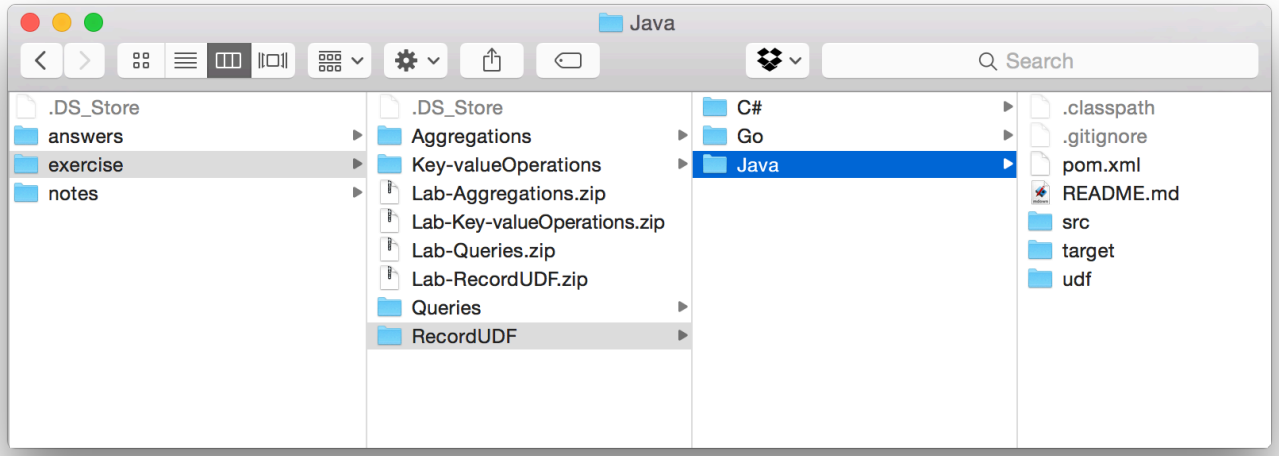
In your cloned or downloaded repository you will find the following directories:

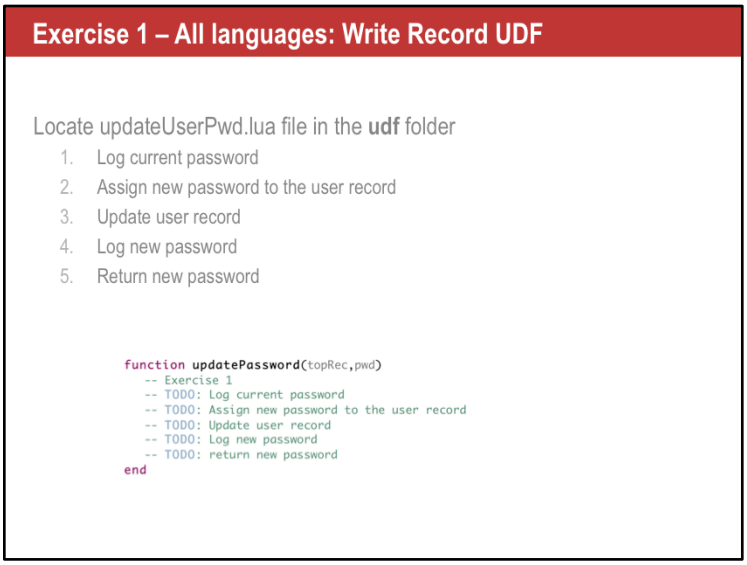
- Answers
- Exercise
- Notes

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.

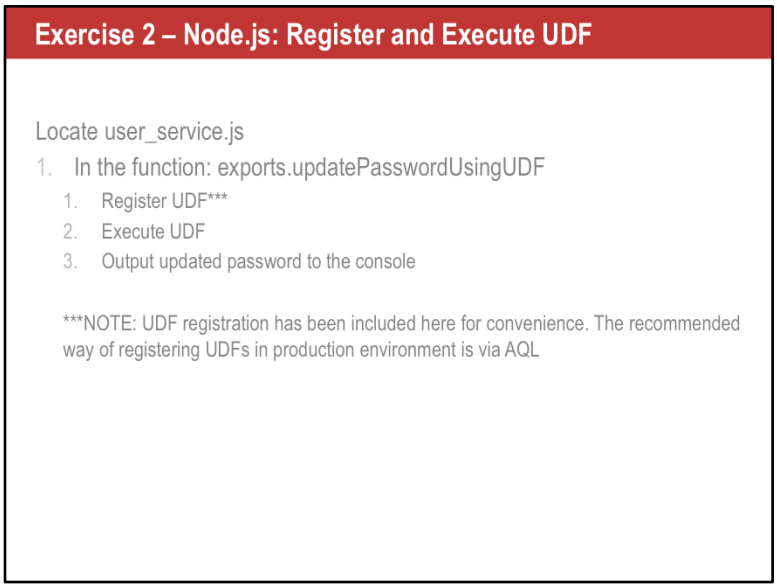




In this exercise you will create a record UDF that:

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 the funcNon module.updatePasswordUsingUDF add your code:

1. Register the UDF with an API call

```
// Register UDF
client.udfRegister('udfs/updateUserPwd.lua', function(err) {
  if ( err.code === 0 ) {
    var UDF = {module:'updateUserPwd', funcname:
      'updatePassword', args: [answers.password]};
    var key = {
      ns:  "test",
      set: "users",
      key: answers.username
    };
```

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

```
    client.execute(key, UDF, function(err) {
      // Check for errors
      if ( err.code === 0 ) {
           console.log("INFO: User password updated successfully!");
      }
      else {
           console.log("ERROR: User password update failed\n", err);
      }
    });
```

3. Output the return from the UDF to the console

```
  } else {
    // An error occurred
     console.error("ERROR: updateUserPwd UDF registeration failed:\n", err);
  }
});
```

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**

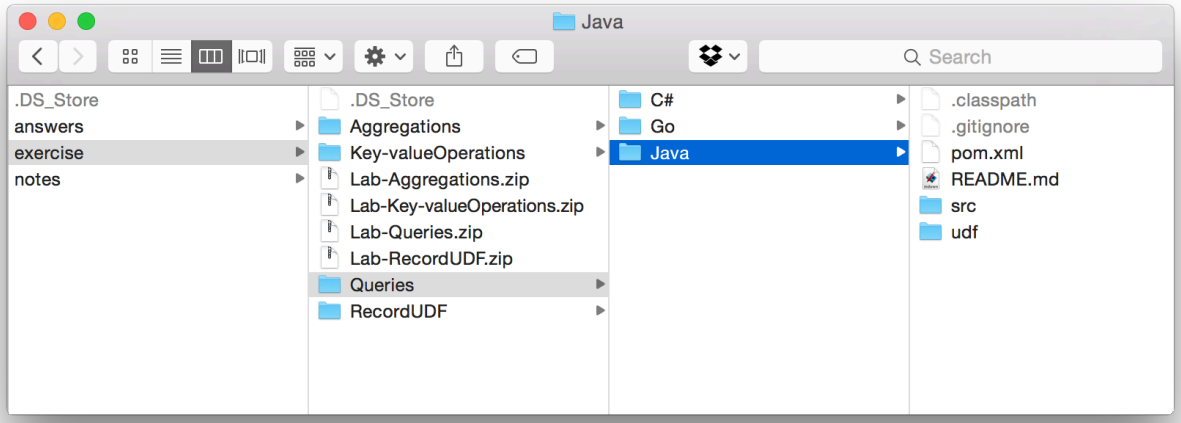
On your cloned or downloaded repository, you will find the following directories:

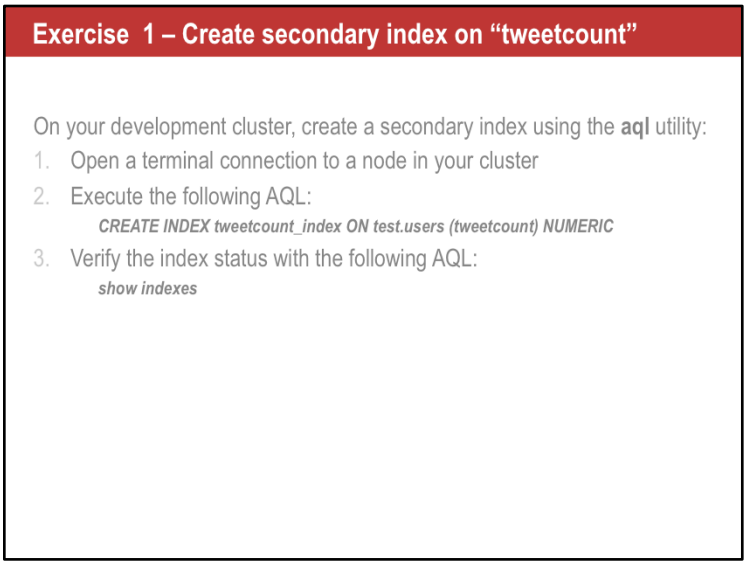
- Answers
- Exercise
- Notes

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.



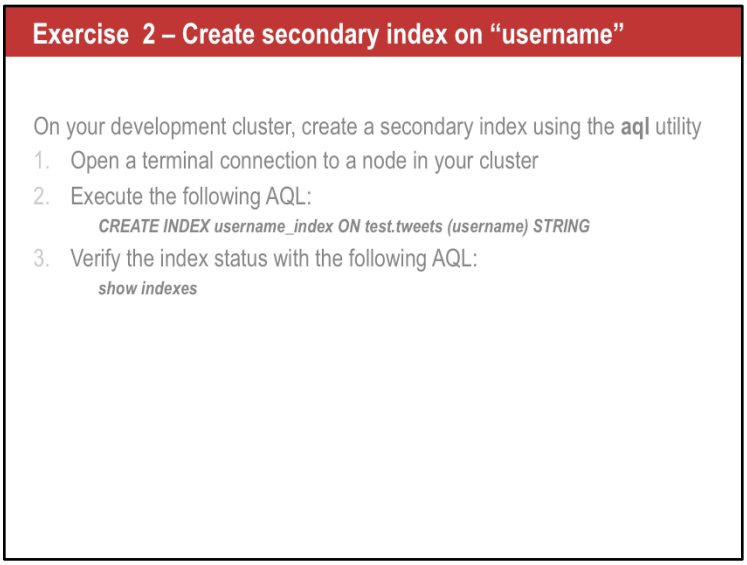


Logon on to your server instance and run **aql** to create a numeric index on *tweetcount*.
At the prompt, enter the command:

CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC

Verify that the index has been created with the command:

show indexes

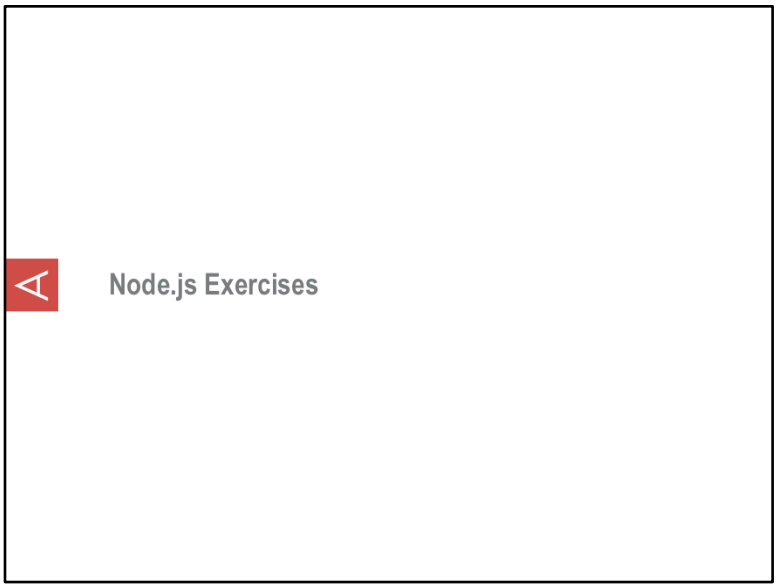


Logon on to your server instance and run **aql** to create a string index on *username*. At the prompt, enter the command:

CREATE INDEX username_index ON test.tweets (username) STRING

Verify that the index has been created with the command:

show indexes


Node.js Exercises

Exercise 3 – Node.js: Query tweets for a given username

Locate tweet_service.js

In tweet_service.js modify the function exports.queryTweetsByUsernamequeryTweetsByUsername(:

1. Create String array of bins to retrieve. In this example, we want to display tweets for a given user.
2. Create Statement instance. On this Statement instance:
   1. Set namespace
   2. Set name of the set
   3. Set equality Filter for username
3. Execute query passing in null policy and instance of Statement created above
4. Process the streamand output tweets to the console

In the funcNon:
queryTweetsByUsername, add your code:

1. Create a statement with
   1. the Namespace
   2. the Set name
   3. the bins (“tweet”)
   4. Set the Filter to qualify the user name

```
var statement = {filters:[aerospike.filter.equal('username',
answer.username)]};
```

2. Execute the query from your code

```
var query = client.query('test', 'tweets', statement);
var stream = query.execute();
```

3. Iterate through the RecordSet returned from the query

```
stream.on('data', function(record)  {
  console.log(record.bins.tweet);
});
stream.on('error', function(err)  {
  console.log('ERROR: Query Tweets By Username failed: ',err);
});
stream.on('end', function()  {
  // console.log('INFO: Query Tweets By Username completed!');
  queryUsersByTweetCount(client);
});
```

Exercise 4 – Node.js: Query users based on tweets

Locate tweet_service.js

In the function: queryUsersByTweetCount:

1. Create Statement instance. On this Statement instance:
   1. Set namespace
   2. Set name of the set
   3. Set array of bins to retrieve
   4. Set range Filter for min--max tweetcount
2. Execute query passing in null policy and instance of Statement created above
3. Process the stream and output text in format "<username> has <#> tweets"

In the funcNon exports.queryUsersByTweetCount, add your code:

1. Create Statement with:
   1. the namespace
   2. the Set
   3. the bins to retrieve
   4. a range Filter for min max tweetcount

```
var statement = {filters:[aerospike.filter.range('tweetcount',
answers.min, answers.max)]};
statement.select = ['username', 'tweetcount'];
```

2. Execute query passing in null policy and instance of Statement created above

```
var query = client.query('test', 'users', statement);
var stream = query.execute();
stream.on('data', function(record)  {
  console.log(record.bins.username + ' === ' +
record.bins.tweetcount);
});
```

3. Process the stream and output text in format "<username> has <#> tweets”

```
stream.on('error', function(err)  {
  console.log('ERROR: Query Users By Tweet Count Range
failed:\n',err);
});
stream.on('end', function()  {
  // console.log('INFO: Query Users By Tweet Count Range
completed!');
});
```

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:

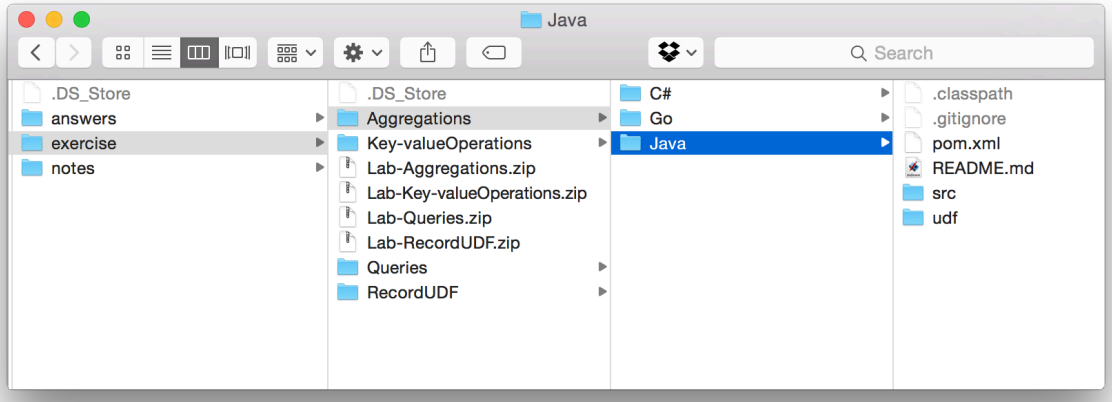
- Answers
- Exercise
- Notes

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
```

Exercise 2 – node.js: Register and Execute UDF

Locate user_service.js

In user_service.js modify the function *exports.aggregateUsersByTweetCountByRegion*

1. Register UDF
2. Create Statement instance. On this Statement instance:
   1. Set namespace
   2. Set name of the set
   3. Set the bins to retrieve
   4. Set min--max range Filter on tweetcount
3. Execute aggregate query passing in <null> policy and instance of Statement, .lua filename of the UDF and lua function name.
4. Process the stream and output result to the console in format "Total Users in <region>: <#>"

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

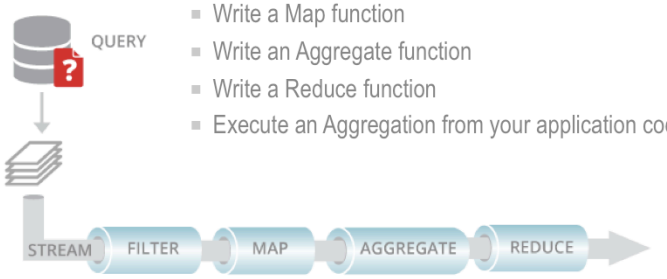
We will programma cally register the UDF for convenience.

In user_service.js modify the funcNon *exports.aggregateUsersByTweetCountByRegion*, add your code to look like this:

1. Register the UDF with an API call
2. Prepare the Statement
3. Execute the query
4. Process the stream

```
// NOTE: UDF registration has been included in here for convenience and to demonstrate the syntax.
// NOTE: The recommended way of creating indexes in production env is via AQL.
//Register UDF
client.udfRegister('udfs/aggregationByRegion.lua', function(err) {
  if ( err.code === 0 ) {
var statement = {filters:[aerospike.filter.range('tweetcount', answers.min, answers.max)],
aggregationUDF: {module: 'aggregationByRegion', funcname: 'sum'}};
    var query = client.query('test', 'users', statement);
    var stream = query.execute();

    stream.on('data', function(result)  {
      console.log('Total Users In East:  ', result.e);
      console.log('Total Users In West:  ', result.w);
      console.log('Total Users In North: ', result.n);
      console.log('Total Users In South: ', result.s);
    });
    stream.on('error', function(err)  {
      console.log('ERROR: Aggregation Based on Tweet Count By Region failed: ',err);
    });
    stream.on('end', function()  {
      console.log('INFO: Aggregation Based on Tweet Count By Region completed!');
    });
  } else {
    // An error occurred
    console.log('ERROR: aggregationByRegion UDF registeration failed: ', err);
  }
});
```

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