



AEROSPIKE

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:

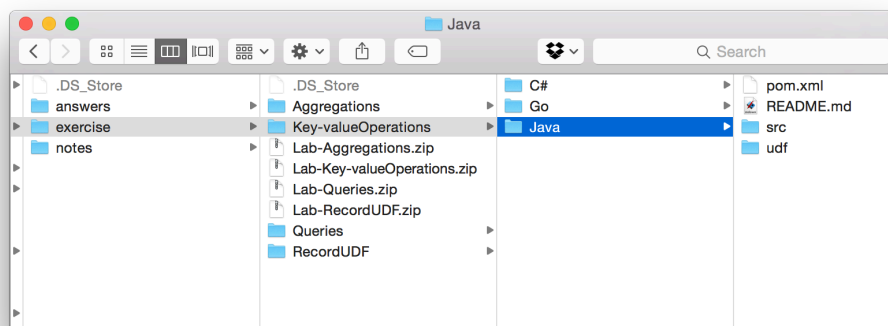
- Answers
- Exercise
- Notes

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

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

The exercises for this module are in the Key-valueOperations directory and you 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



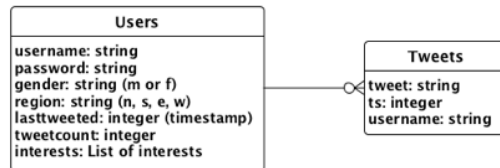
Tweetaspike Data Model

Users

- Namespace: test, Set: users, Key: <username>
- Bins:
 - username - String
 - password - String (For simplicity password is stored in plain-text)
 - gender - String (Valid values are 'm' or 'f')
 - region - String (Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) -- to keep data entry to minimal we just store the first letter)
 - lasttweeted - int (Stores epoch timestamp of the last/most recent tweet) -- Default to 0
 - tweetcount - int (Stores total number of tweets for the user) -- Default to 0
 - interests - Array of interests

Tweets

- Namespace: test, Set: tweets, Key: <username:<counter>>
- Bins:
 - tweet - string
 - ts - int (Stores epoch timestamp of the tweet)
 - 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
region	String	Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) -- to keep data entry to minimal we just store the first letter
lasttweeted	Integer	Stores epoch timestamp 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 timestamp of the tweet
username	String	User name of the tweeter



Java Exercises

Exercise 1 – Java: Connect & Disconnect

Locate Program Class in the Maven project

1. Create an instance of `AerospikeClient` with one initial IP address. Ensure that this connection is created only once
2. Add code to disconnect from the cluster. Ensure that this code is executed only once

```
// TODO: Create new AerospikeClient instance
```



In this exercise you will connect to a Cluster by creating an `AerospikeClient` 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 `AerospikeClient` 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 constructor for the class `Program`, add code similar to this;

```
// Establish a connection to Aerospike cluster  
this.client = new AerospikeClient("192.168.1.15", 3000);
```

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

2. At the end of method `work()`, 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.

```
if (client != null && client.isConnected()) {  
    // Close Aerospike server connection  
    client.close();  
}
```

Exercise 2 – Java: Write Records

Create a User Record and Tweet Record

Locate UserService and TweetService classes in the Maven project

1. Create a User Record – In UserService.createUser()
 1. Create an instance of WritePolicy
 2. Create Key and Bin instances for the User Record
 3. Write User Record
2. Create a Tweet Record – In TweetService.createTweet()
 1. Create an instance of WritePolicy
 2. Create Key and Bin instances for the Tweet Record
 3. Write Tweet Record
 4. Update Tweet count and last Tweeted timestamp in the User Record

Create a User Record. In UserService.createUser(), add code similar to this:

1. Create a WritePolicy

```
// Write record
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
```

2. Create Key and Bin instances

```
Key key = new Key("test", "users", username);
Bin bin1 = new Bin("username", username);
Bin bin2 = new Bin("password", password);
Bin bin3 = new Bin("gender", gender);
Bin bin4 = new Bin("region", region);
Bin bin5 = new Bin("lasttweeted", 0);
Bin bin6 = new Bin("tweetcount", 0);
Bin bin7 = new Bin("interests",
    Arrays.asList(interests.split(",")));
```

3. Write a user record using the Key and Bins

```
client.put(wPolicy, key, bin1, bin2, bin3, bin4,
    bin5, bin6, bin7);
```

Create a Tweet Record. In TweetService.createTweet(), add code similar to this:

1. Read user record

2. Create a WritePolicy

```
// Check if username exists
userKey = new Key("test", "users", username);
userRecord = client.get(null, userKey);

WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
```

3. Create Key and Bin instances

```
tweetKey = new Key("test", "tweets", username + ":"
    + nextTweetCount);
Bin bin1 = new Bin("tweet", tweet);
Bin bin2 = new Bin("ts", ts);
Bin bin3 = new Bin("username", username);
```

1. Write a tweet record using the Key and Bins

```
client.put(wPolicy, tweetKey, bin1, bin2, bin3);
console.printf("\nINFO: Tweet record created!\n");
```

2. Update the user record with tweet count.

Note: Also add code in updateUser() method.

```
// Update tweet count and last tweet'd timestamp in the user
// record
long ts = getTimeStamp();
updateUser(client, userKey, wPolicy, ts, nextTweetCount);
```


Exercise 2 ...Cont.– Java: Read Records

Create a User Record, Tweet Record and then a Read User Record

Locate UserService and TweetService classes in the Maven project

1. Read User record – In UserService.getUser()
 1. Read User Record
 2. Output User Record to the console

Read a User Record. In UserService.getUser(), add code, similar to this, to:

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

```
// Check if username exists
userKey = new Key("test", "users", username);
userRecord = client.get(null, userKey);
if (userRecord != null) {
    console.printf("\nINFO: User record read successfully! Here are the details:\n");
    console.printf("username:  " + userRecord.getValue("username") + "\n");
    console.printf("password:  " + userRecord.getValue("password") + "\n");
    console.printf("gender:    " + userRecord.getValue("gender") + "\n");
    console.printf("region:    " + userRecord.getValue("region") + "\n");
    console.printf("tweetcount: " + userRecord.getValue("tweetcount") + "\n");
    console.printf("interests:  " + userRecord.getValue("interests") + "\n");
} else {
    console.printf("ERROR: User record not found!\n");
}
```

Exercise 3 – Java: Batch Read

Batch Read Tweets for a given user

Locate UserService class in the Maven project

1. In UserService.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 UserService.batchGetUserTweets(), add code similar to this:

1. Read a user record

```
userKey = new Key("test", "users", username);
userRecord = client.get(null, userKey);
```

2. Get the tweet count

```
// Get how many tweets the user has
int tweetCount = userRecord.getInt("tweetcount");

// Create an array of keys so we can initiate batch read
// operation
```

3. Create a "list" of tweet keys

```
Key[] keys = new Key[tweetCount];
for (int i = 0; i < keys.length; i++) {
    keys[i] = new Key("test", "tweets",
        (username + ":" + (i + 1)));
}
console.printf("\nHere's " + username + "'s tweet(s):\n");
```

4. Perform a Batch operation to read all the tweets

```
// Initiate batch read operation
```

5. Then print out the tweets

```
if (keys.length > 0){
    Record[] records = client.get(new BatchPolicy(), keys);
    for (int j = 0; j < records.length; j++) {
        console.printf(records[j].getValue("tweet").toString() + "\n");
    }
}
```

Exercise 4 – Java: Scan

Scan all tweets for all users

Locate TweetService class in the Maven project

1. In TweetService.scanAllTweetsForAllUsers()
 1. Create an instance of ScanPolicy
 2. Set policy parameters (optional)
 3. Initiate scan operation that invokes callback for outputting tweets to the console

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

In the TweetService.scanAllTweetsForAllUsers(), add code similar to this:

1. Create a ScanPolicy
2. Set policy parameters
3. Perform a Scan operation and process the results

```
// Java Scan
ScanPolicy policy = new ScanPolicy();
policy.concurrentNodes = true;
policy.priority = Priority.LOW;
policy.includeBinData = true;
```

```
client.scanAll(policy, "test", "tweets", new ScanCallback() {

    public void scanCallback(Key key, Record record)
        throws AerospikeException {

        console.printf(record.getValue("tweet") + "\n");

    }
}, "tweet");
```

Exercise 5 – Java: Read-modify-write

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

Locate UserService class in the Maven project

1. In UserService.updatePasswordUsingCAS()
 1. Create a WritePolicy
 2. Set WritePolicy.generation to the value read from the User record.
 3. Set WritePolicy.generationPolicy to EXPECT_GEN_EQUAL
 4. Update the User record with the new password using the GenerationPolicy

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

In UserService.updatePasswordUsingCAS(), add code similar to this:

1. Create a WritePolicy
2. Set WritePolicy.generation to the value read from the User record.
3. Set WritePolicy.generationPolicy to EXPECT_GEN_EQUAL
4. Update the User record with the new password using the GenerationPolicy

```
// Check if username exists
userKey = new Key("test", "users", username);
userRecord = client.get(null, userKey);
if (userRecord != null)
{
    // Get new password
    String password;
    console.printf("Enter new password for " + username + ":");
    password = console.readLine();

    WritePolicy writePolicy = new WritePolicy();
    // record generation
    writePolicy.generation = userRecord.generation;
    writePolicy.generationPolicy = GenerationPolicy.EXPECT_GEN_EQUAL;
    // password Bin
    passwordBin = new Bin("password", Value.get(password));
    client.put(writePolicy, userKey, passwordBin);

    console.printf("\nINFO: The password has been set to: " + password);
}
```

Exercise 6 – Java: Operate

Update Tweet count and timestamp and examine the new Tweet count

Locate TweetService class in the Maven project

1. In TweetService.updateUser()
 1. Comment out code added in Exercise 2 for updating tweet count and timestamp
 2. Uncomment line updateUserUsingOperate(client, userKey, policy, ts, tweetCount);
 3. In updateUserUsingOperate(client, userKey, policy, ts, tweetCount)
 1. Initiate operate passing in policy, user record key, .add operation incrementing tweet count, .put operation updating timestamp and .get operation to read the user record
 2. Output updated Tweet count to console

Aerospike can perform multiple operations on a record in one transaction. Update the tweet count and timestamp in a user record and read the new tweet count.

In TweetService.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
// updateUserUsingOperate(client, userKey, policy, ts);
```

1. In TweetService.updateUserUsingOperate(), add code similar to this:

```
Record record = client.operate(policy, userKey,
    Operation.add(new Bin("tweetcount", 1)),
    Operation.put(new Bin("lasttweeted", ts)),
    Operation.get());
```

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](#)

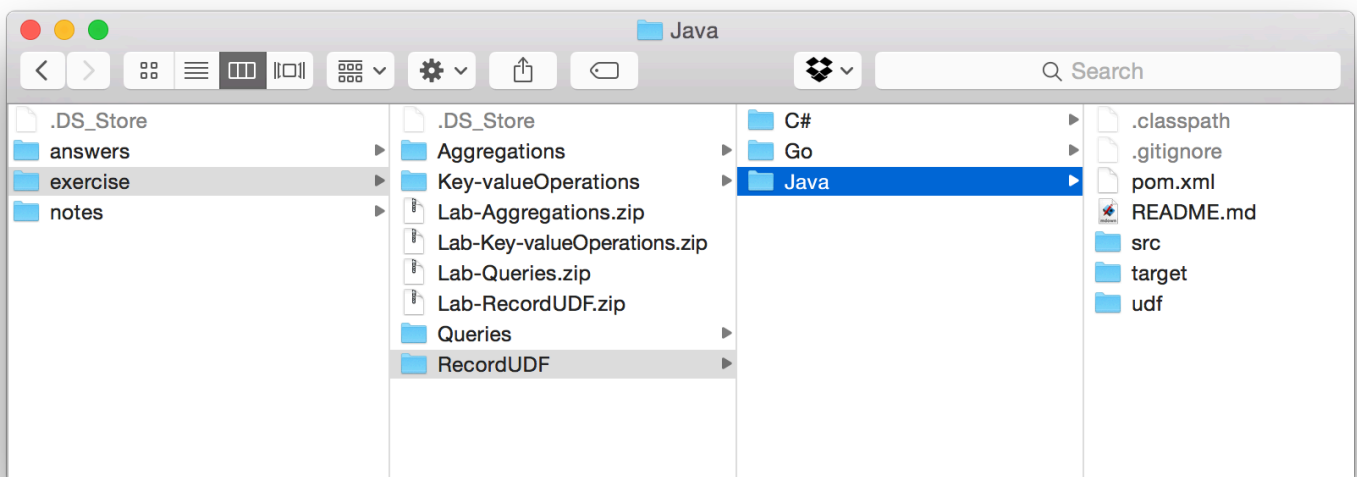
In your cloned or downloaded repository you will find the following directories:

- Answers
- Exercise
- Notes

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

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

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



Exercise 1 – All languages: Write Record UDF

Locate updateUserPwd.lua file in the **udf** folder

1. Log current password
2. Assign new password to the user record
3. Update user record
4. Log new password
5. Return new password

```
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
```

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
```

Exercise 2 – Java: Register and Execute UDF

Locate UserService class

1. In UserService.updatePasswordUsingUDF()
 1. Register UDF***
 2. Execute UDF
 3. Output updated password to the console

***NOTE: UDF registration has been included here for convenience. The recommended way of registering UDFs in production environment is via AQL

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

We will programmatically register the UDF for convenience.

In UserService.updatePasswordUsing UDF(), locate these comments and add your code:

1. Register the UDF with an API call

```
LuaConfig.SourceDirectory = "udf";
File udfFile = new File("udf/updateUserPwd.lua");
RegisterTask rt = client.register(null, udfFile.getPath(),
                                   udfFile.getName(), Language.LUA);
rt.waitTillComplete(100);
```

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

```
String updatedPassword = (String) client.execute(null, userKey,
                                                  "updateUserPwd", "updatePassword",
                                                  Value.get(password));
```

1. Output the return from the UDF to the console

```
console.printf("\nINFO: The password has been set to: " + updatedPassword);
```

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](#)

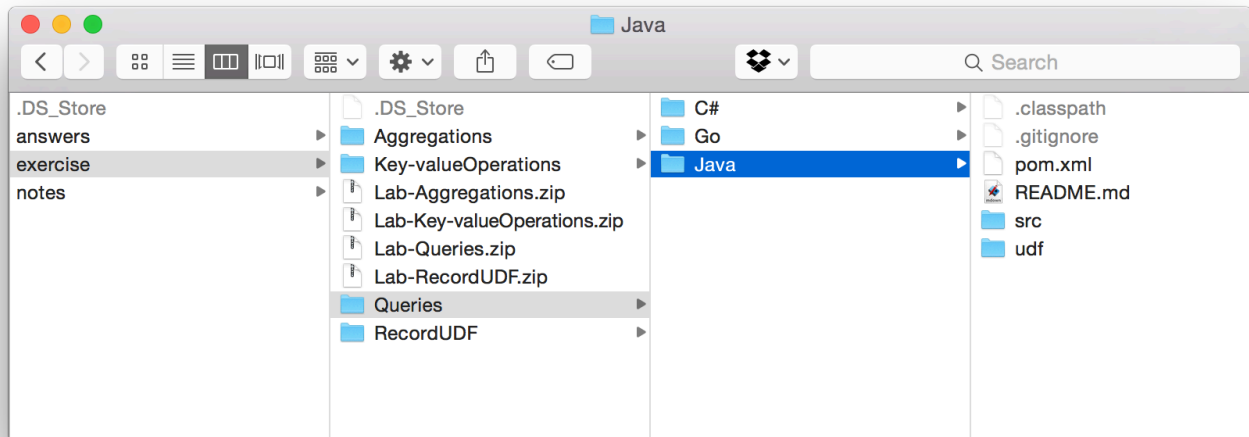
On your cloned or downloaded repository, you will find the following directories:

- 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 you will find a Project/Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.



Exercise 1 – Create secondary index on “tweetcount”

On your development cluster, create a secondary index using the **aql** utility:

1. Open a terminal connection to a node in your cluster
2. Execute the following AQL:

```
CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC
```
3. Verify the index status with the following AQL:

```
show indexes
```

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
```


Exercise 2 – Create secondary index on “username”

On your development cluster, create a secondary index using the **aql** utility

1. Open a terminal connection to a node in your cluster
2. Execute the following AQL:

```
CREATE INDEX username_index ON test.tweets (username) STRING
```
3. Verify the index status with the following AQL:

```
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
```



Java Exercises

Exercise 3 – Java: Query tweets for a given username

Locate class TweetService in the Maven project

In TweetService.queryTweetsByUsername():

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 array of bins to retrieve
 4. Set equality Filter for username
3. Execute query passing in null policy and instance of Statement created above
4. Iterate through returned ResultSet and output tweets to the console

In TweetService.queryTweetsByUsername(),
locate these comments and add your code:

1. Create a list of Bins to retrieve
 2. Create a statement
 1. Set the Namespace
 2. Set the Set name
 3. Set the index name (optional)
 4. Set the array of bins (from above)
 5. Set the Filter to qualify the user name
 3. Execute the query from your code
 4. Iterate through the ResultSet returned from the query
 5. Close the record set
- ```
if (username != null && username.length() > 0) {
 String[] bins = { "tweet" };

 Statement stmt = new Statement();
 stmt.setNamespace("test");
 stmt.setSetName("tweets");
 stmt.setIndexName("username_index");
 stmt.setBinNames(bins);
 stmt.setFilters(Filter.equal("username", username));

 console.printf("\nHere's " + username + "'s tweet(s):\n");

 rs = client.query(null, stmt);
 while (rs.next()) {
 Record r = rs.getRecord();
 console.printf(r.getValue("tweet").toString() + "\n");
 }
} else {
 console.printf("ERROR: User record not found!\n");
}
} finally {
 if (rs != null) {
 // Close record set
 rs.close();
 }
}
```

## Exercise 4 – Java: Query users based on number of tweets

Locate class TweetService in AerospikeTraining Solution

In TweetService.queryUsersByTweetCount():

1. Create String array of bins to retrieve. In this example, we want to output which user has how many tweets.
2. 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
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"

In TweetService.queryUsersByTweetCount(),  
locate these comments and add your code:

1. Create String array of bins to retrieve. In this example, we want to output which user has how many tweets.
2. Create Statement instance. On this Statement instance:
  1. Set namespace
  2. Set name of the set
  3. Set name of the index
  4. Set array of bins to retrieve
  5. Set range Filter for min--max tweetcount
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"
5. Close the RecordSet

```
String[] bins = { "username", "tweetcount", "gender" };
```

```
Statement stmt = new Statement();
stmt.setNamespace("test");
stmt.setSetName("users");
stmt.setIndexName("tweetcount_index");
stmt.setBinNames(bins);
stmt.setFilters(Filter.range("tweetcount", min, max));
```

```
rs = client.query(null, stmt);
```

```
while (rs.next()) {
 Record r = rs.getRecord();
 console.printf(r.getValue("username") + " has "
 + r.getValue("tweetcount") + " tweets\n");
}
} finally {
 if (rs != null) {
 // Close record set
 rs.close();
 }
```

## 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, Ruby, 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:

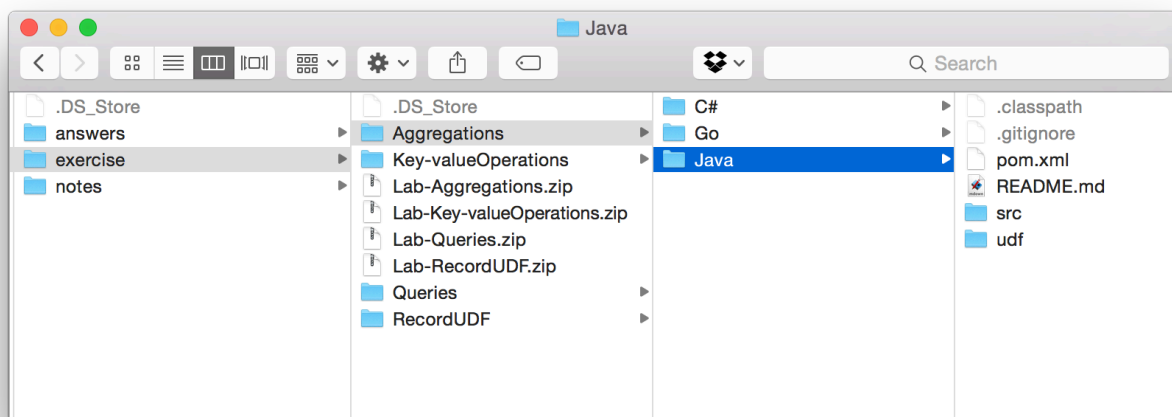
- Answers
- Exercise
- Notes

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 Aggregations directory and you 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**





## 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() function is invoked one for each element in the stream.
- Reduces the aggregations into a single Map of values – The reduce\_stats() function is invoked once for each data partition, once for each node in the cluster, and finally once on the client.
- The sum() function configures the stream processing, and it is the function 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 – Java: Register and Execute UDF

Locate UserService class in the Maven project

In UserService.aggregateUsersByTweetCountByRegion()

1. Register UDF
2. Create String array of bins to retrieve. In this example, we want to display total users that have tweets between min-max range by region.
3. 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 min--max range Filter on tweetcount
4. Execute aggregate query passing in the Statement, UDF module and function name.
5. Examine returned ResultSet 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 programmatically register the UDF for convenience .

In UserService.aggregateUsersByTweetCountByRegion(), add your code to look like this:

1. Register the UDF with an API call

```
// NOTE: UDF registration has been included here for convenience and to demonstrate the syntax.
// The recommended way of registering UDFs in production env is via AQL
LuaConfig.SourceDirectory = "udf";
File udfFile = new File("udf/aggregationByRegion.lua");

RegisterTask rt = client.register(null, udfFile.getPath(),
 udfFile.getName(), Language.LUA);
rt.waitTillComplete(100);
```
2. Create the Bin array
3. Prepare the Statement

```
String[] bins = { "tweetcount", "region" };
Statement stmt = new Statement();
stmt.setNamespace("test");
stmt.setSetName("users");
stmt.setBinNames(bins);
stmt.setFilters(Filter.range("tweetcount", min, max));
rs = client.queryAggregate(null, stmt, "aggregationByRegion", "sum");
if (rs.next()) {
 Map<Object, Object> result = (Map<Object, Object>) rs
 .getObject();
 console.printf("\nTotal Users in North: " + result.get("n") + "\n");
 console.printf("Total Users in South: " + result.get("s") + "\n");
 console.printf("Total Users in East: " + result.get("e") + "\n");
 console.printf("Total Users in West: " + result.get("w") + "\n");
}
} finally {
 if (rs != null) {
 // Close result set
 rs.close();
 }
}
```
4. Execute the aggregation
5. Examine the ResultSet
6. Close the result set

## 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