Aggregations

Goals

This module describes how to use Aggregations. At the end of this module you will be able to :

- Use a Filter operation
- Use a Map operation
- Use an Aggregate operation
- Use a Reduce operation
- Execute a Stream UDF from your application

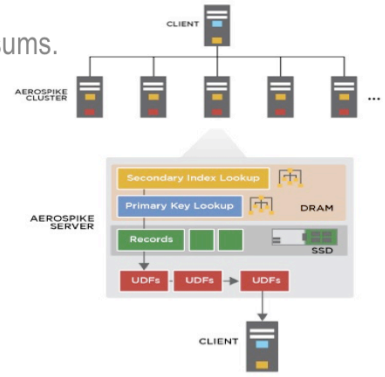
Aggregations

Programmatic framework similar to **MapReduce**. Processes a collection of rows based on something like a "where" clause.

Primarily used for counts, aggregate and sums.

An Aggregation

- Runs in parallel on all cluster nodes
- Lua for function language
- Indexed – **prequalifies** records
- Reduce in the client



Aggregations

The Aggregation framework is a programmatic framework is similar to a MapReduce system, in that an initial Map function is run over a collection, and emits results in a highly parallel fashion. Those results flow as a stream through a pipeline of either subsequent map steps, reduction steps, and aggregation steps. The simple use case is counts and aggregate sums inside the database.

Indexed MapReduce

One of the main differences from other systems in that the aggregation is done against an index - essentially a WHERE clause. By filtering against an index performance can be very high.

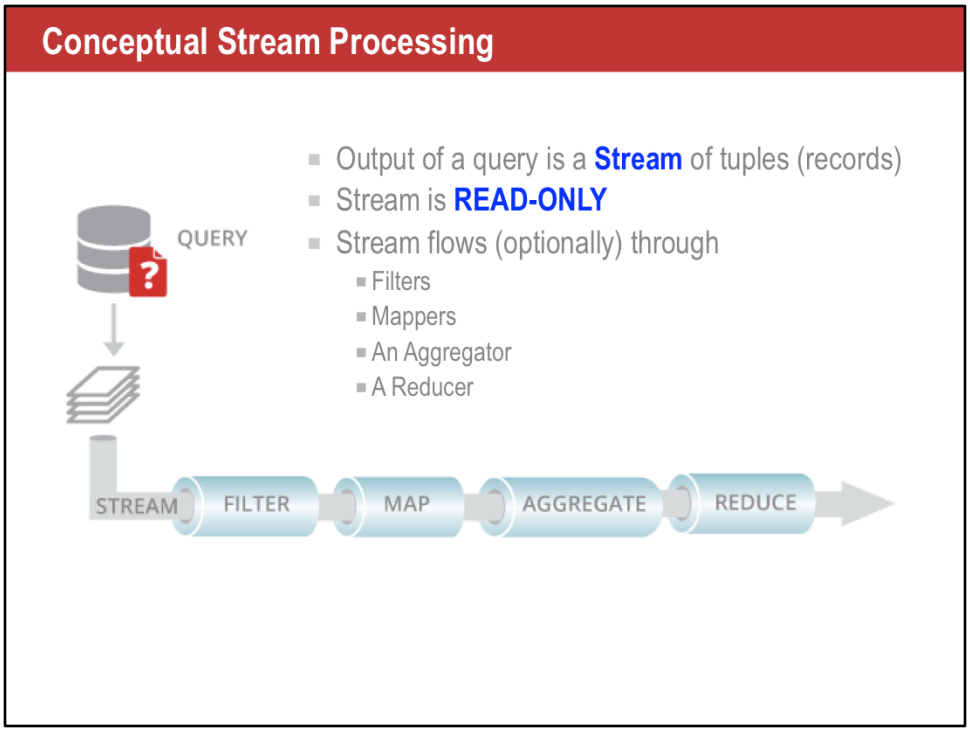
The aggregation system is implemented using User Defined Functions (UDFs) written in Lua. Functionality written in C can be called from Lua.

Each client sends the aggregation request to all the servers in the cluster, who processes results independently, and sends individual results back to the requesting client. The client then runs a final reduce phase, also in Lua, to sum the results.

Use Cases

The Aerospike Labs example Real-time analytics with Aerospike shows code, and a real-world dataset, which determines which airline had the greatest number of late flights in January 2012.

By having a secondary index on a bin with the update timestamp, an Aggregation can quickly gather statistics on records that have recently changed. Compared to standard MapReduce systems, which act on an entire dataset without indexes, Aerospike Aggregation can touch fewer records.



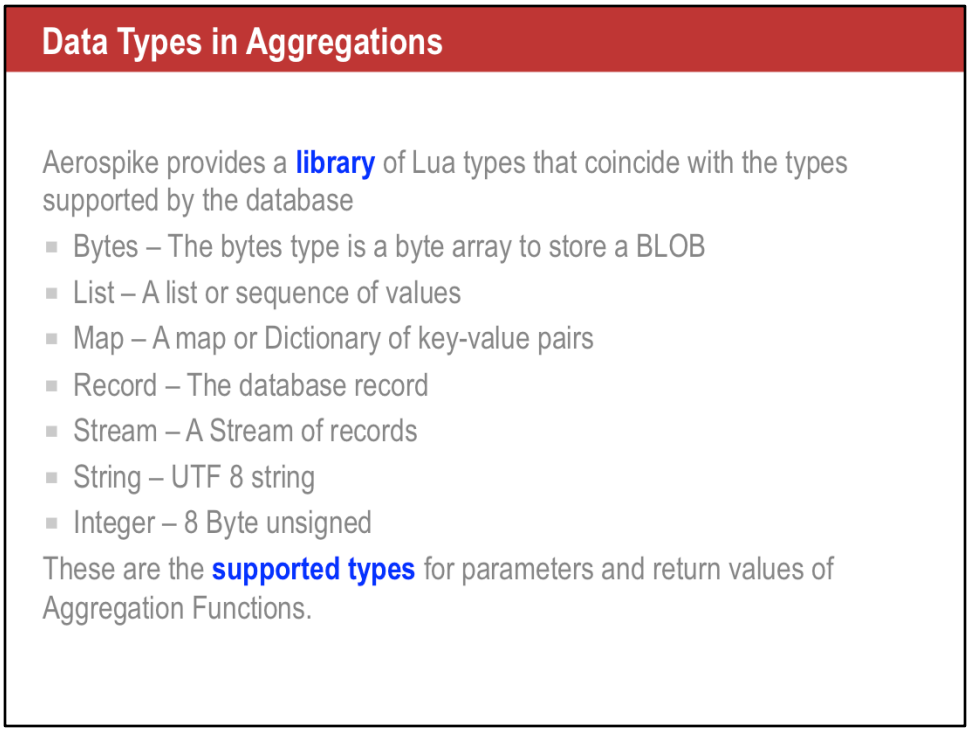
Conceptual stream processing

Consider the output of a query as tuples (records) flowing in a stream. The stream is READ-ONLY.

The contents of the stream could simply flow to the client as a standard result of the query, but by adding any number of aggregation functions the stream can be:

- **Filter** – One or more filters applied to "filter out" tuples that are not required. A filter function decides if the tuple is allowed to continue in the stream, or be removed from the stream.
- **Map** – One or more map function(s) are used to "transform" the data in a tuple.
- **Aggregate** – The aggregate function aggregates a stream of data info a single value.
- **Reduce** – Reducing is the gathering of intermediary results and reducing them into the final output results. The reduce function run on each node, reducing the output from that node, and the final reduce is executed on the client, reducing the data gathered from each node.

You can have more that one kind of Aggregation Function in the stream.



Data Types in Aggregations

Aerospike provides a library of Lua types that coincide with the types supported by the database. These are the basic type that can be used as parameters and return values.

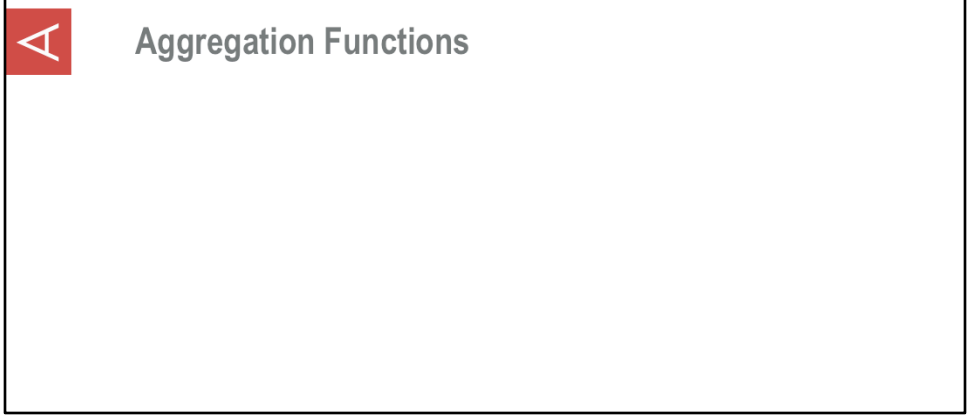
Bytes - The bytes type provides the ability to build a byte array using bytes and **Integer -** This type coincides with BLOB type in the database.

List - A list is data structure that represents a sequence of values.

Map — A collection of (key, value) pairs, in which a key can only appear once in the collection.

Record — Represents database records, including bins – (name, value) pairs – and metadata.

Stream - Represents streams of records.


Aggregation Functions

Function Stereotypes

Aerospike used UDFs to to implement the stream operations. There are four basic function stereotypes:

- Filter
- Map
- Aggregate
- Reduce

Filter Operation

The filter operation filters values from the stream

Parameter:

- Tuple

Output:

- Boolean


```
local function my_filter_fn(record)
     if record['age'] > 18 and record['sex'] == 'male' then
         return true
     else
         return false
     end
end
```

Filter Operation

The filter operation will filter values from the stream. The filter operation accepts a single argument, the filter function, e.g.

return s : filter(my_filter1).

Filter Function

The filter function accepts the current value from the stream and should return true or false, where true indicates the value should continue down the stream.

A typical filter function looks as below. A query will feed records in to the stream. The filter operation will apply the filter function for each record in the stream. In the example, the filter function allows records containing "males" older than "18" years to be passed down the stream.

```
local function my_filter_fn(record)
     if record['age'] > 18 and record['sex'] == 'male' then
         return true
     else
         return false
     end
end
```

Zero or more filter functions can be configured to process the stream. This allows a modular, and generalized, filter design. Your can construct a library of filters and

Map Operation

The map operation will **transform** values in the stream, by accepting a value from the stream, and replacing it with a new value

Parameter:

- Current stream element

Output:

- New stream element


```
local function my_map_fn(rec)
     local result = map()
     result['name'] = rec['name']
     result['city'] = rec['city']
     return result
end
```

Map Operation

The map operation transforms values in the stream. Function signature:

return s : map(my_map1).

Map Function

The map function accepts a value from the stream, and returns a value which will be passed to the next function in the processing chain.

The type of the return value must be one of those supported by the database (see Slide 5)

A simple example map function

```
local function my_map_fn(rec)
     local result = map()
     result['name'] = rec['name']
     result['city'] = rec['city']
     return result
end
```

In this example you can see that only “name” and “city” are returned to the stream.

A Map function can also filter, this is often done in normal MapReduce.

Aggregate Operation

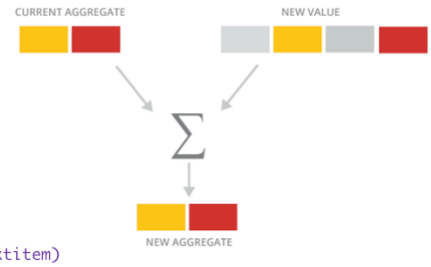
Aggregate operation aggregates a stream of data into a single aggregate value by taking **current** aggregate, and the **next** value from the stream, to produce a **composite** aggregate.

Parameters:

- Current aggregate
- Next stream element

Output:

- Composite aggregate




```
local function my_aggregation_fn(aggregate, nextitem)
     -- If the count for the city does not exist, initialize it with 1
     -- Else increment the existing counter for that city
     aggregate[nextitem['city']] = (aggregate[nextitem['city']] or 0) + 1
     return aggregate
end
```

The Aggregate Operation

Aggregate operation aggregates a stream of data into a single aggregate value. The aggregate operation function two arguments and returns one value.

The arguments are

- the aggregate value
- the next value from the input stream.

The function should return a single value that is the aggregate of the current aggregate and the value from the stream, thereby forming a new aggregate The return type must be one of those supported by the database. (Slide 5)

The aggregate value and the return value should be of same type. The most efficient approach is to aggregate value passed in and combine it with the next value from the input stream.

A typical aggregate function looks as below. This example is doing a “group-by” type of operation where it is calculating the number of citizens in each city.

```
local function my_aggregation_fn(aggregate, nextitem)
     -- If the count for the city does not exist, initialize it with 1
     -- Else increment the existing counter for that city
     aggregate[nextitem['city']] = (aggregate[nextitem['city']] or 0) + 1
     return aggregate
end
```

TIP: The aggregate function takes a collection elements from its input stream (across multiple invocations) and will return only one element to its output stream. There is little benefit putting two aggregate functions in a row, because the output of the first aggregate function will only emit single element which can be consumed by the next aggregate function.

The accumulated aggregate value can grow quite large. It is possible, for example, to simulate the SQL Select DISTINCT function by accumulating values in a map and then dumping the map at the end.

Reduce Operation

The reduce operation will reduce values in the stream to a single value.

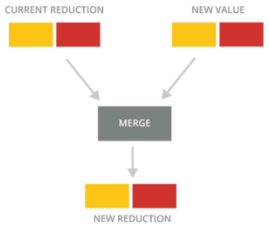
- Executes on Cluster side and Client side

Parameters

- Cumulative reduction
- Next stream element

Output

- New cumulative Reduction




```
local function my_merge_fn(val1, val2)
     return val1 + val2
end

local function my_reduce_fn(global_agg_value, next_agg_value)
     -- map.merge is a library function will call the my_merge_fn
     -- for each key and returns a new map.
     -- my_merge_fn will be passed the values of the key in two maps as arguments.
     -- It stores the result of merge function against the key in the result map
     return map.merge(global_agg_value, next_agg_value, my_merge_fn)
end
```

The Reduce Operation

The reduce operation will reduce values in the stream to a single value. The reduce operation accepts a single argument, the **reduce function**.

The reduce function

The reduce function accepts two values from the stream and returns a single value that is fed back into the function as the first argument. The reduce function should be cumulative for the best performance. The two arguments of the reduce function and the return value should be of the same type. The type of the return value must be one of those supported by the database (Slide 5).

One main characteristic of reduce function is that it executes both on the server nodes as well as the client side (in application instance). Each node first runs the data stream through the functions defined in the stream definition. The end result of this is sent to the application instance. The application gets results from all the nodes in the cluster. The client layer in it does the final reduce using the reduce function specified in the stream. So, the reduce function should be able to accumulate the intermediate values (coming form the cluster nodes).

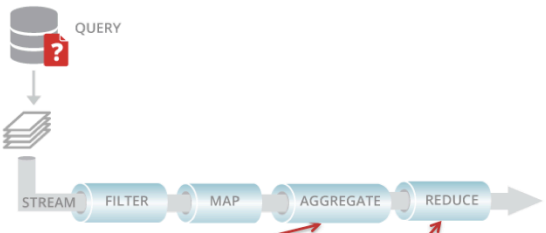
If there is no reduce function, the client layer is passes all the data coming from the nodes to the application.

Stream Function

A Stream function **configures** the stream processing. This is the function that you execute from the client. It is:

- Registered with the cluster

```
local function aggregate_stats(map,rec)
  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 map
end
local function reduce_stats(a,b)
  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 a
end
function sum(stream)
   return stream : aggregate(map{n=0,s=0,e=0,w=0},aggregate_stats) : reduce(reduce_stats)
end
```



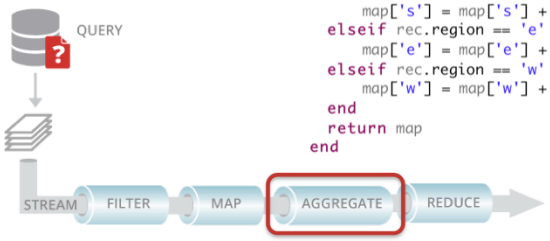
Aggregate Configuration

```
aggregate(map{n=0,s=0,e=0,w=0},aggregate_stats)
```

Initial aggregate: map() initalized with 4 keys: n,s,e,w

Aggregate function

```
local function aggregate_stats(map,rec)
  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 map
end
```



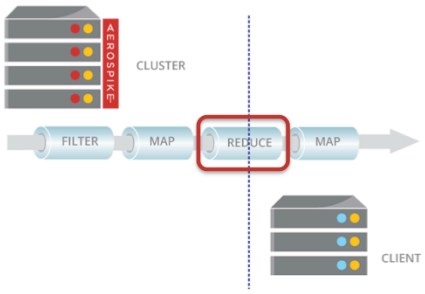
Reduce Configuration

The Reduce function configured as the final stage of the stream processing

```
reduce(reduce_stats)

local function reduce_stats(a,b)
  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 a
end
```

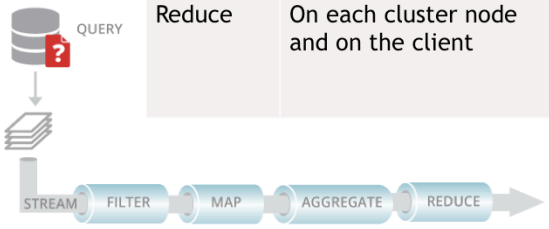
Reduce function



The Reduce function is configured as the final stage of the stream processing. The reduce function is run on each ode in the cluster and on the client.

Function Invocation

Function	Where	When
Filter	On same cluster node as the element	Each element in the stream
Map	On same cluster node as the element	Each element in the stream
Aggregate	On same cluster node as element	Each element in the stream
Reduce	On each cluster node and on the client	On each node for each element in the stream. On the client for each node in the cluster



Function invocation

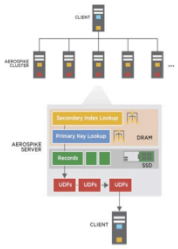
Stream functions are invoked on the node where the stream element is produced.

The Filter, Map and Aggregate functions are executed on the node on the same node as the stream element, and as the element is produced (from a query or previous stage of the stream).

The Reduce function is invoked on each cluster node, for element in the stream, and on the client for each node in the cluster.

Prepare and Execute an Aggregation

Prepare and execute a query, and invoke the Stream UDF:



```
// C# prepare and execute
string[] bins = { "username", "tweetcount", "gender", "region" };
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.QueryAggregate(null, stmt, "aggregation_region", "sum");
```

```
// Java prepare and execute
String[] bins = { "username", "tweetcount", "gender", "region" };
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.queryAggregate(null, stmt, "aggregation_region", "sum");
```

Module

Function

Processing the Results

Results are returned in a **ResultSet** object (collection).

- Process by iterating
- Valid Types:
  - String
  - Integer
  - List
  - Map
  - Bytes

```
// C# process results
ResultSet rs = client.QueryAggregate(null, stmt, "aggregation_region", "sum");
if (rs.Next())
{
      Dictionary<object, object> result = (Dictionary<object, object>)rs.Object;
      Console.WriteLine("Total Users in North: " + result["n"]);
      Console.WriteLine("Total Users in South: " + result["s"]);
      Console.WriteLine("Total Users in East: " + result["e"]);
      Console.WriteLine("Total Users in West: " + result["w"]);
}

// Java process results
ResultSet rs = client.queryAggregate(null, stmt, "aggregation_region", "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");
}
```

Processing the results

The Aggregate operation returns a ResultSet object. It is similar to, but not the same as, a RecordSet. You iterate over this collection to retrieve the output of the final stage of your stream, usually the Reduce function.

The valid types of the elements in the ResultSet are: String, Integer, List and Map, which translate directly to the equivalent C# or Java types. This example's final stage is a Reduce function that produces a Map. This translates to a Dictionary in C# or a Map in Java.

Example aggregation: Average

In this simple **Average** example we average the the total number of Tweets between a range, by getting the sum and count.

```
// Java averages example
Statement stmt = new Statement();
stmt.setNamespace("test");
stmt.setSetName("users");
stmt.setFilters(Filter.range("l1", 0, 1000));
ResultSet rs = client.queryAggregate(null, stmt, "average_example", "average");

-- Lua averages module
function average(s)

    local function accumulate(out, rec)
        out['sum'] = (out['sum'] or 0) + (rec['l1'] or 0)
        out['count'] = (out['count'] or 0) + 1
        return out
    end

    local function reducer(a, b)
        local out = map()
        out['sum'] = a['sum'] + b['sum']
        out['count'] = a['count'] + b['count']
        return out
    end

    return s : aggregate(map{sum = 0, count = 0}, accumulate) : reduce(reducer)
end
```

Average Example

In this simple **Average** example we are averaging the the total number of tweets between a range, by getting the sum and count.

Average Module

The stream function “average” operates on the Bin “l1”. The function is configured with a:

- Aggregate operation implemented by “accumulate”. It accumulates the ‘sum’ value and the ‘count’ value in a Map. Note: this map is created once and add to on each invocation of the “accumulate” function.
- Reduce operation implemented by “reducer”. Simply adds the ‘sum’ values and ‘count’ values returns a Map containing these two values.

The Map returned the client can the be used to perform a simple average calculation.

Remember:

Make your functions lean

The Aggregate operation is invoked once for **every record in the output** of the query.

The Reduce operation is invoked for: every data partition, every node in the cluster, and once on the client.


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# 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 on your USB stick or in your "unzipped" directory

```
~/exercise/Aggregations/<language>
```

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

On your USB stick, or in your "unzipped" directory, you will find the following directories:

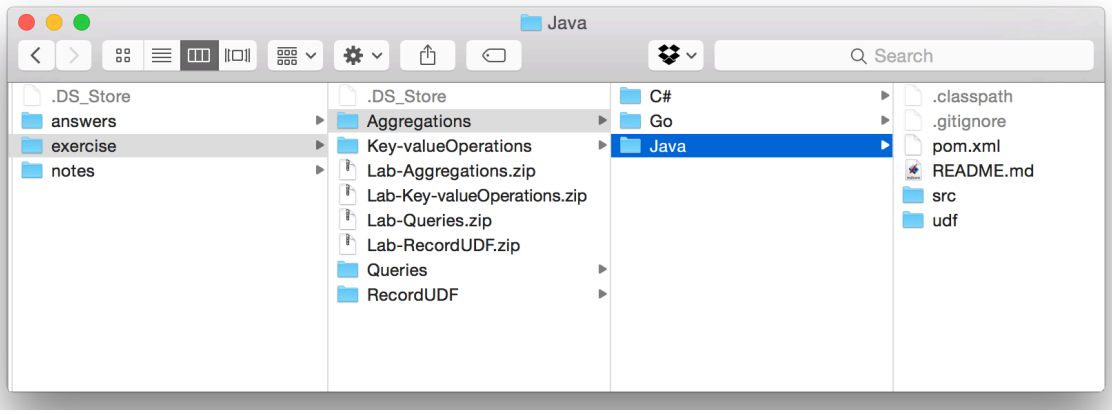
- Answers
- Exercise
- Notes

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

- C#
- Java

The exercises for this module are in the Aggregations 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



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 AerospikeTraining Solution

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 name of the index
    4. Set array of bins to retrieve
    5. Set min--max range Filter on tweetcount
4. Execute aggregate query passing in <null> policy and instance of Statement, .lua filename of the UDF and lua function name.
5. Examine returned ResultSet and output result to the console in format "Total Users in <region>: <#>"

***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 – **NEVER** do this in production code.

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

1. Register the UDF with an API call
2. Create the Bin array
3. Prepare the Statement
4. Execute the aggregation
5. Examine the ResultSet
6. Close the result set

```
// 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);

     String[] bins = { "tweetcount", "region" };
     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.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();
```

Exercise 2 – C#: Register and Execute UDF

Locate UserService class in AerospikeTraining Solution

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 name of the index
   4. Set array of bins to retrieve
   5. Set min--max range Filter on tweetcount
4. Execute aggregate query passing in <null> policy and instance of Statement, .lua filename of the UDF and lua function name.
5. Examine returned ResultSet and output result to the console in format "Total Users in <region>: <#>"

***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 – **NEVER** do this in production code.

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
string luaDirectory = @"..\..\udf";
LuaConfig.PackagePath = luaDirectory + @"\?.lua";
string filename = "aggregationByRegion.lua";
string path = Path.Combine(luaDirectory, filename);
RegisterTask rt = client.Register(null, path, filename, Language.LUA);
rt.Wait();
```

2. Create the Bin array
3. Prepare the Statement

```
string[] bins = { "tweetcount", "region" };
Statement stmt = new Statement();
stmt.SetNamespace("test");
stmt.SetSetName("users");
stmt.SetIndexName("tweetcount_index");
stmt.SetBinNames(bins);
stmt.SetFilters(Filter.Range("tweetcount", min, max));

Console.WriteLine("\nAggregating users with " + min + "-" + max + " tweets by region. Hang on...\n");
```

4. Execute the aggregation

```
rs = client.QueryAggregate(null, stmt, "aggregationByRegion", "sum");
```

5. Examine the ResultSet

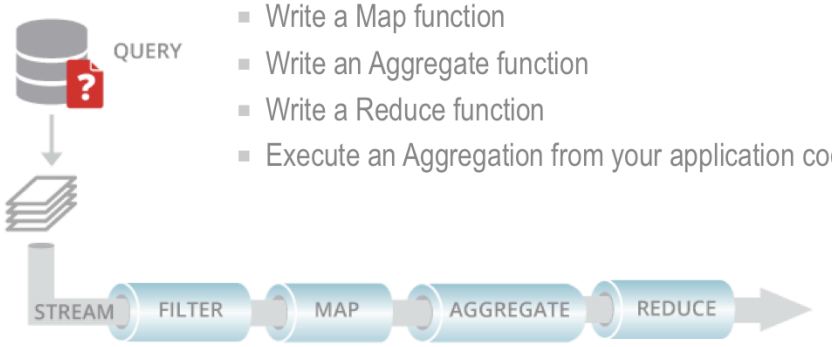
```
if (rs.Next())
{
    Dictionary<object, object> result = (Dictionary<object, object>)rs.Object;
    Console.WriteLine("Total Users in North: " + result["n"]);
    Console.WriteLine("Total Users in South: " + result["s"]);
    Console.WriteLine("Total Users in East: " + result["e"]);
    Console.WriteLine("Total Users in West: " + result["w"]);
}
}
finally
{
    if (rs != null)
```

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