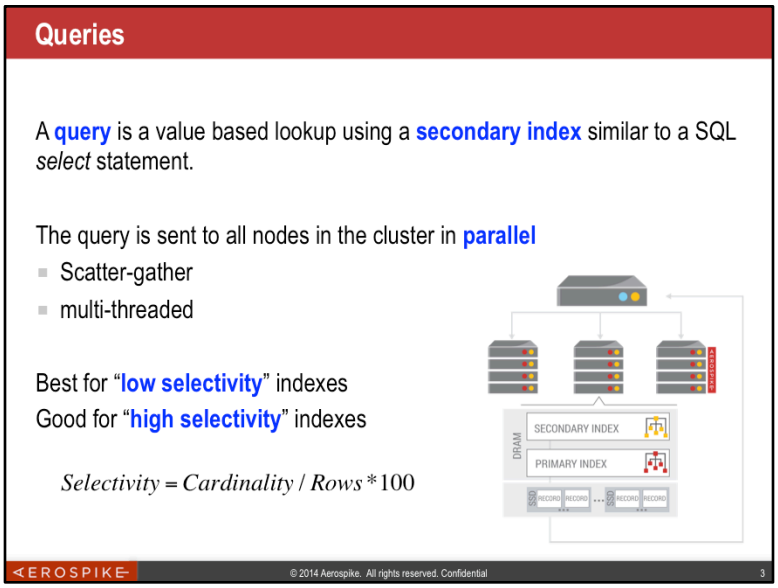
Queries

Objectives

A the the end of this module you will be able to:

- Define and manage secondary indexes
- Execute a query on a secondary index in
  - C#
  - Java
  - Go
  - PHP
- Process the results of a query
- Correctly handle errors



Query

Aerospike query provides value-based look up through the use of secondary indexes. Best suited for high cardinality queries. Query result returned as set of records, like a select statement in SQL.

The query is sent to all nodes in parallel in a scatter-gather pattern. The query is scattered. Worker threads in the client gather the results from all the nodes in the cluster and the results are returned as a handle to a Record Set

Cardinality and Selectivity

Cardinality is the number of unique values. For example the population of Europe is approx. 744 million but the cardinality based on male and female is 2.

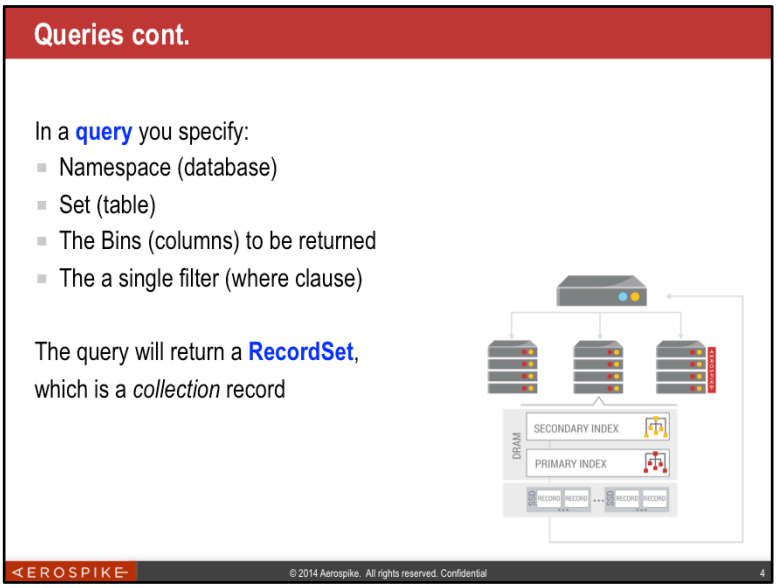
Selectivity of an index is the cardinality divided by the number of rows and expressed as a percentage.

$$Selectivity = Cardinality / Rows * 100$$

Therefore the Selectivity of the Europe population based gender is: 0.00003%

Conversely the Selectivity of the European population based on Last Name would be much higher.

Aerospike is best where the number records in the results is in the range of 1k to 1 million.



Query

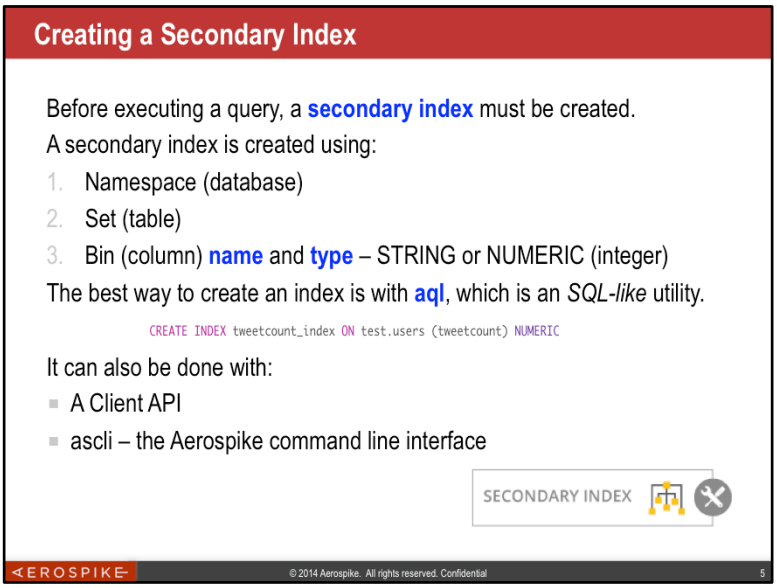
You specify the Namespace, Set, Bins you want to retrieve and a single predicate filter (a were clause) in a Statement and then use that Statement in the Query() method.

The results are returned as a handle to a RecordSet. The Record Set is a collection that can be iterated over to retrieve each record.

The "null" set

The **null** set is a set with no name. Records without a Set name are stored in the **null** set. In a query, the **null** set is treated like any other Set.

Note: You must have sufficient memory (heap space) in your application to accommodate the records returned in the RecordsSet. Aerospike does not support cursors yet.



Creating an Index

An index consumes RAM for every index entry, and background index creation can take a substantial amount of resources. Index creation should be scheduled carefully on an operational system. An Index is created using:

1. Namespace (database)
2. Set (table) including the null Set
3. Bin name AND type (column)

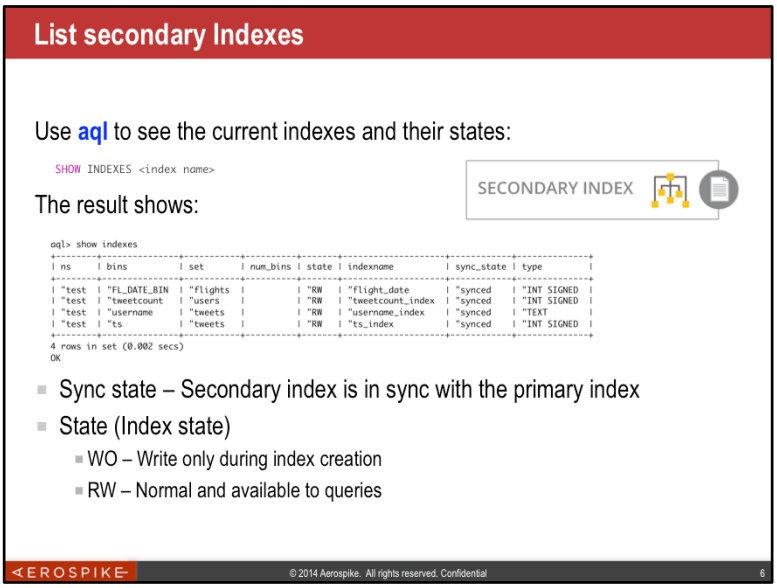
Remember: a named Bin can have a different type in different records – No Schema.

The best way to create and manage indexes in an Aerospike cluster is using the **aql** tool.

```
CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC
```

aql is officially distributed as part of the server tools package. It is a Linux command line tool that implements Aerospike Query Language (AQL) which is a SQL like language. A java based **aql** implementation and an Eclipse plugin is available at Aerospike Labs http://www.aerospike.com/community/labs/

You can also create indexes with the client API. Creating indexes takes time and resources and should **NOT be done in your production code**.



Showing an Index

The best way to see the details of an index is by using the **aql** command: show index <namespace>. The namespace is optional and if omitted the indexes for all name spaces are shown.

Index Sync State

Each secondary index has a value called sync_state, which specifies whether the secondary index is in sync with the primary index.

- synced - The secondary index is in sync with the primary index.
- need_sync - The secondary index may not be in sync with the primary index.

Index State

Each secondary index has a value called state,

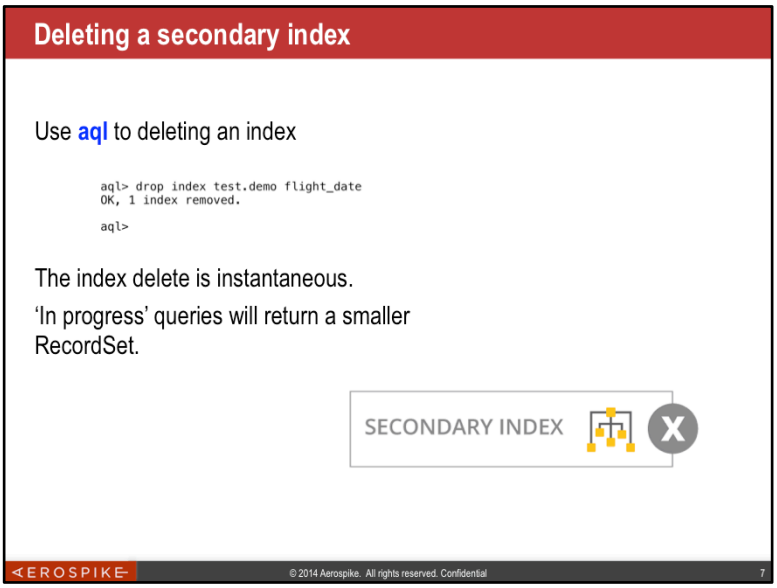
- WO - Secondary index in Write Only Mode. Normal Aerospike put will update secondary index but queries cannot be performed.
- RW - Secondary index in Read Write Mode. Normal and available to Queries.

A secondary index should be in RW state on all the nodes before query can use it.

Repairing an Index

When an index has sync_state=need_sync and state=RW, then it may need to be repaired. **aql** does not provide the ability to repair indexes. For the time being, you will need to use Aerospike Info (asinfo) to execute a repair, on each node in the cluster.

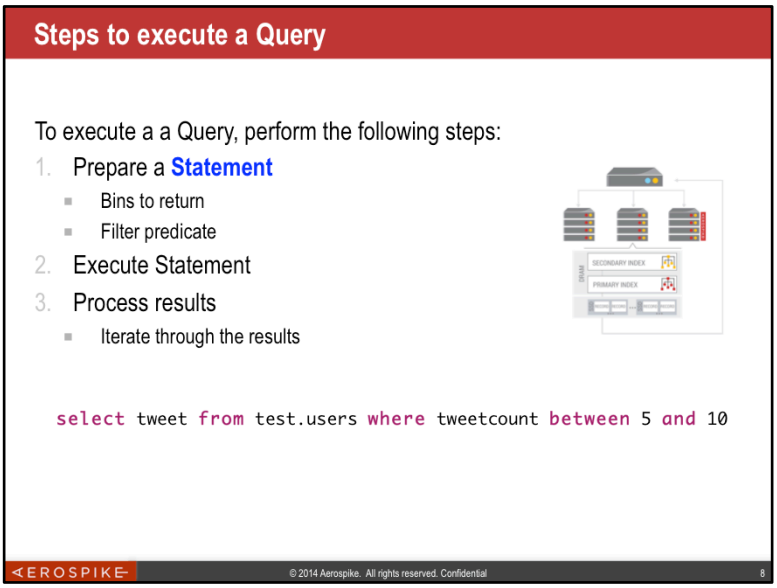
```
$ asinfo -v "sindex-repair:ns=test;indexname=ind_name;set=set_name;"
```



Deleting a secondary index.

The index is deleted instantaneously, memory in each node associated with the index is released and the index definition is removed.

Note: Deleting a secondary index is not atomic, and if Queries are using it they will return a smaller record set than expected



Steps to executing a query

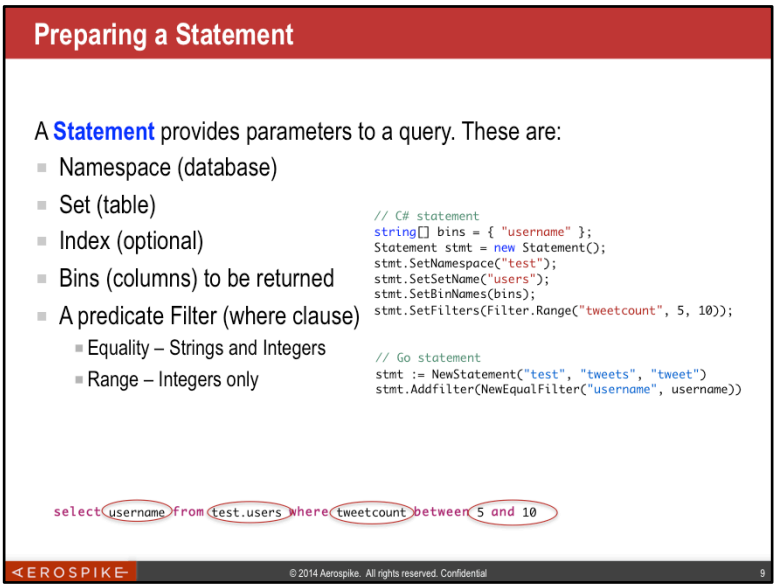
To execute a query you first need to prepare a Statement by supplying a list of Bins to be returned, and supplying a Filter (predicate) that constrains the query.

Consider this **aql** example:

```
select tweet from test.users where tweetcount between 5 and 10
```

To code the equivalent, you should follow these steps:

1. Prepare a Statement containing the “tweet” Bin, and a range filter on the “tweetcount” Bin who’s value is between 5 and 10 (inclusive)
2. Execute the statement with a Query operation – the query can have an optional Query Policy to modify the default behavior. The Query operation returns a RecordSet collection.
3. Process the results in the RecordSet by integrating through the collection.



Preparing a Statement

A Statement is a data structure that provides parameters to the query. In Java and C# it is an object instance.

A Statement works with a specific Namespace and Set (including the null set) and cannot be used to query across Sets (joins).

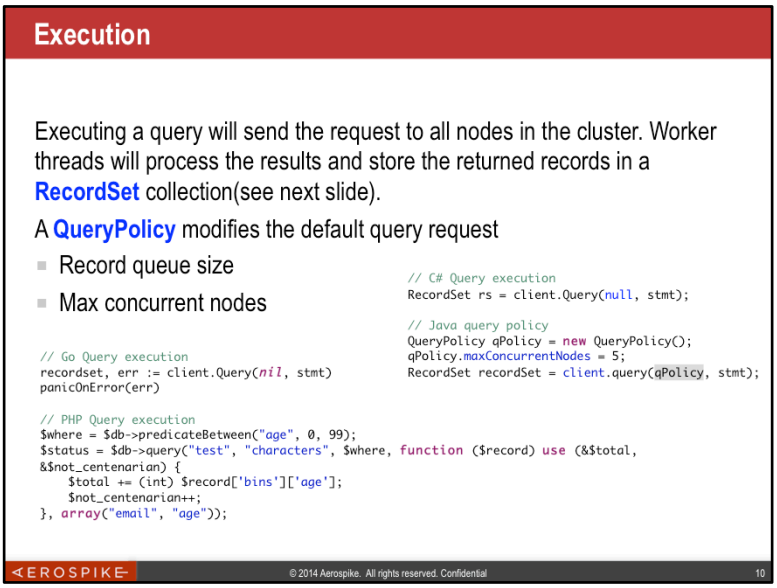
A collection of Bin names is optional and will determine the Bins returned.

Filters

Only one filter valid and it is used to qualify the query. The filter can be:

- Equality filter – equivalent to the SQL “where tweetcount = 5”. Equality filters can be used with Integer and String Bin types.
- Range filter – equivalent to the SQL “where tweetcount between 5 and 10”. Range filters can be used only with Integer bin.

After a Statement is prepared, it can be used as many times as you want.



Execute a Query

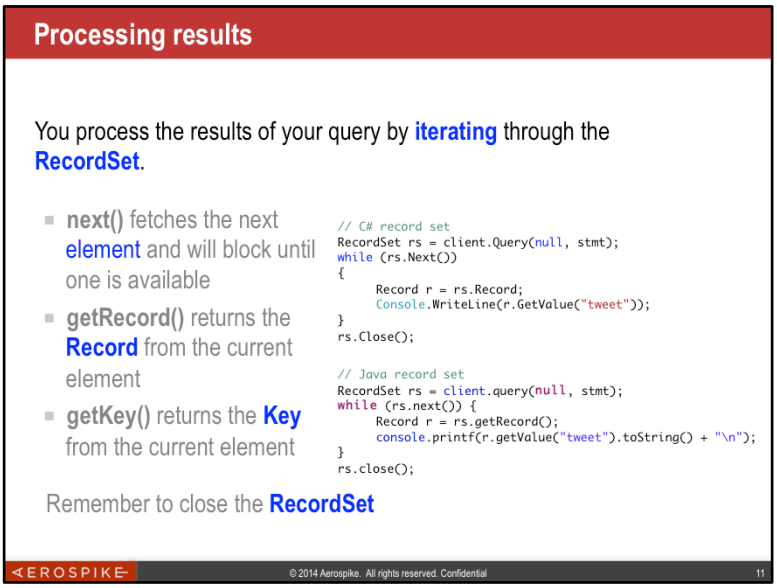
By default, the query execution will send (scatter) the query to all nodes in the cluster and will be executed in parallel. Each node will and return records to the the client. The records are made available to the application in a **RecordSet** collection. The query executor uses a separate worker thread for each node and puts the returned records on a queue internal to the RecordSet. The application thread can concurrently pop records off the queue through the record iterator.

Query Policy

Query policy modifies the default query request.

- Max concurrent nodes - Maximum number of concurrent requests to server nodes at any point in time. If there are 16 nodes in the cluster and max concurrent nodes is 8, then queries will be made to 8 nodes in parallel. When a query completes, a new query will be issued until all 16 nodes have been queried. Default (0) is to issue requests to all server nodes in parallel.
- Record queue size - Number of records to place in queue before blocking. Records received from multiple server nodes will be placed in a queue. A separate thread consumes these records in parallel. If the queue is full, the producer threads will block until records are consumed.

Recommendation: Use the defaults by omitting the QueryPolicy



Processing results

A **RecordSet** is a collection you can iterate over to retrieve the queried records. Each element in the collection will contain a **Key** and a **Record**. Elements in the **RecordSet** are not ordered (no "orderby" constraint), and are not limited. So if your query selects 1 million records, 1 million records will be in your **RecordSet**. This is an important memory consideration. Internally, the **RecordSet** is implemented as a queue. Elements are placed in the queue as they are returned from a cluster node and they are removed from the queue when **next()** is called. The internal queue is thread safe.

Each element in the collection contains a **Record** and a **Key**.

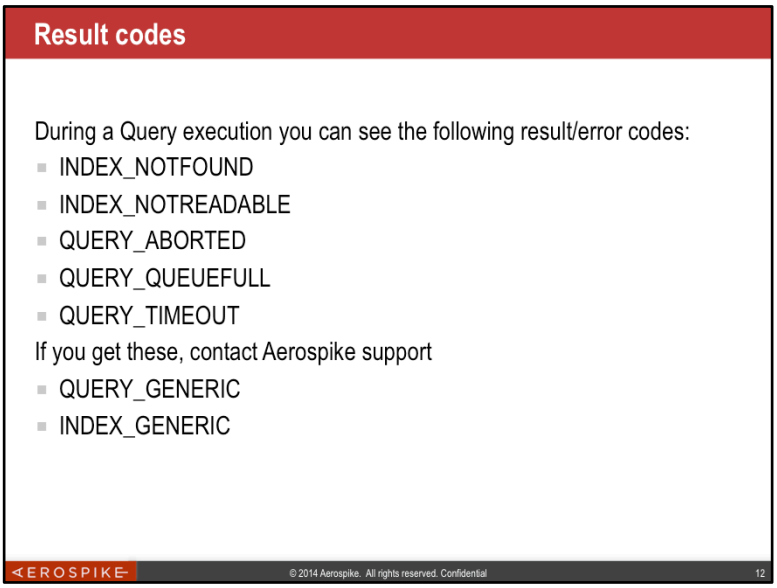
The **next()** method returns a Boolean and fetches the element from the queue and makes it available to be used.

The **getKey()** method returns a **Key** object, because the Key object is returned from the server it will contain a Digest and not the value of the primary key.

The **getRecord()** method returns a **Record** object. This object will contain the generation count along with the requested bins.

A common way to process a **RecordSet** is to use a "while" loop with the Boolean condition being the return from the next() method. When you complete the processing of the **RecordSet**, remember to close it to flag it as as being consumed.

Note: If you request 1 million records, you will need to have memory (heap space) for them, and you should de-reference the **RecordSet** as soon as possible to allow garbage collection to free the heap space.



In addition to the result codes from standard operations, Queries have specific result code relating to secondary indexes and the query operation.

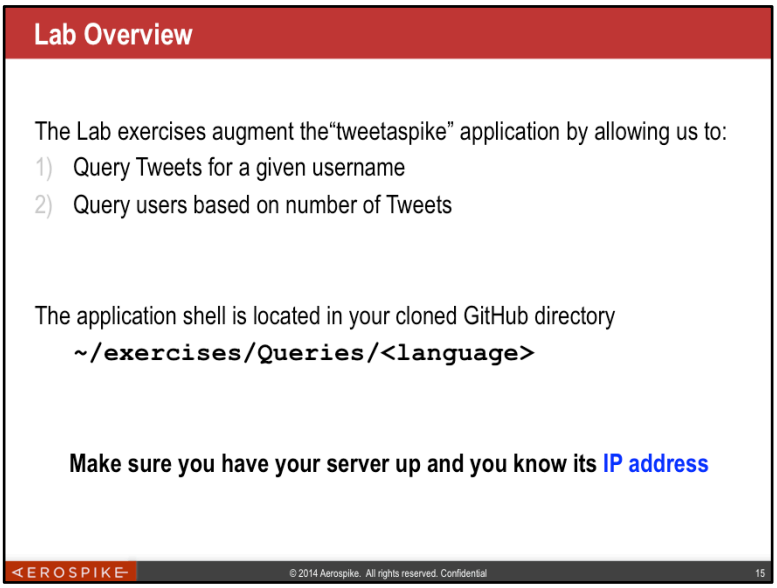
Code	Symbolic name	Description
200	INDEX_FOUND	Secondary index already exists.
201	INDEX_NOTFOUND	Requested secondary index does not exist.
201	INDEX_OOM	Secondary index memory space exceeded.
203	INDEX_NOTREADABLE	Secondary index not available.
204	INDEX_GENERIC	Generic secondary index error. Call Aerospike support.
205	INDEX_NAME_MAXLEN	Index name maximum length exceeded.
206	INDEX_MAXCOUNT	Maximum number of indices exceeded.
210	QUERY_ABORTED	Secondary index query aborted.
211	QUERY_QUEUEFULL	Secondary index queue full.
212	QUERY_TIMEOUT	Secondary index query timed out on server.
213	QUERY_GENERIC	Generic query error. Call Aerospike support.


Lab: Queries
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
13

Objectives

After successful completion of this Lab module you will have:

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



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

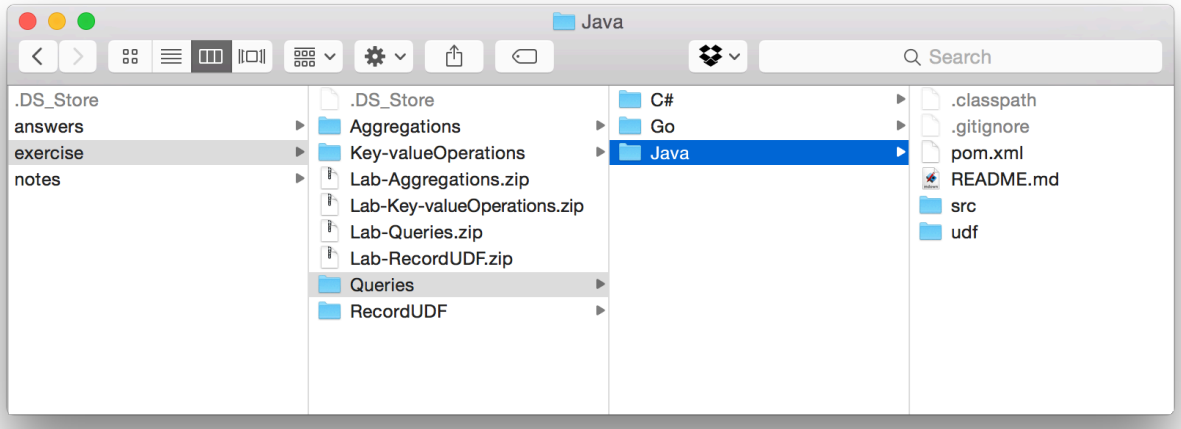
- Answers
- Exercise
- Notes

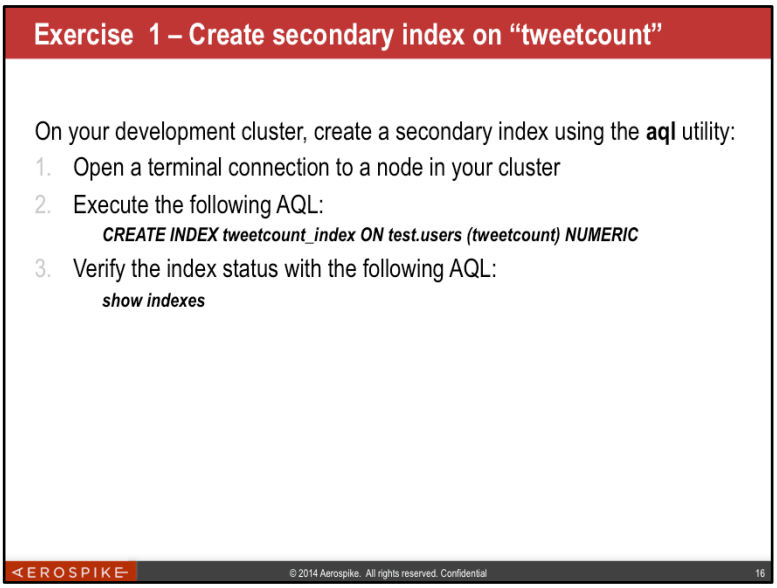
In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Go

The exercises for this module are in the Queries 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





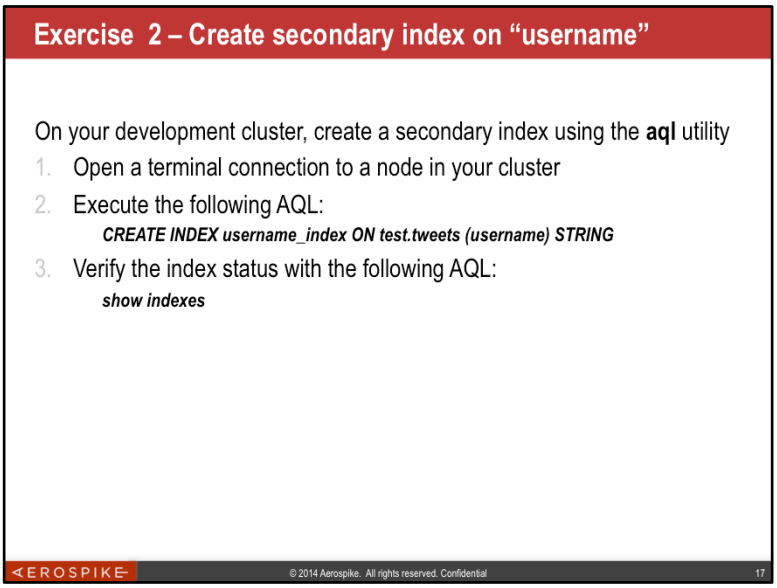
Logon on to your server instance and run **aql** to create a numeric index on *tweetcount*.

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


Java Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
18

Exercise 3 – Java: Query tweets for a given username

Locate class TweetService in the Maven project

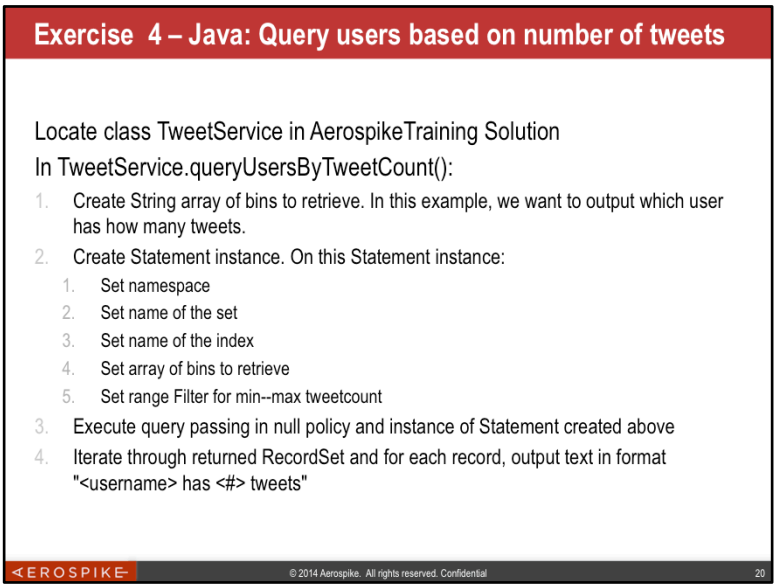
In TweetService.queryTweetsByUsername():

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 RecordSet 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 RecordSet 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();
```



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.

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

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

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

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

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

4. Iterate through returned RecordSet and for each record, output text in format "<username> has <#> tweets"

```
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();
```

5. Close the RecordSet


C# Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
21

Exercise 3 – C#: Query tweets for a given username

Locate class TweetService in AerospikeTraining Solution

In TweetService.queryTweetsByUsername():

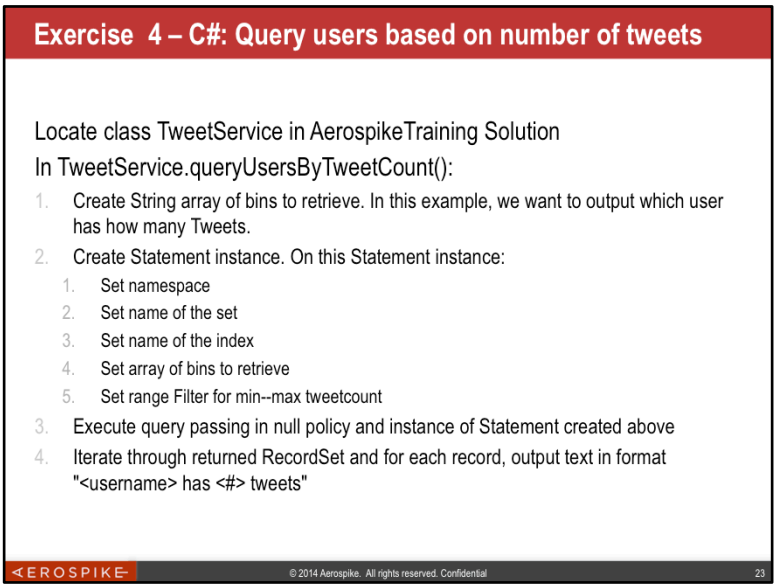
1. Create String array of bins to retrieve. In this example, we want to display tweets for a given user.
2. Create Statement instance. On this Statement instance:
   1. Set namespace
   2. Set name of the set
   3. Set name of the index
   4. Set array of bins to retrieve
   5. Set equality Filter for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

AEROSPIKE 22

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 RecordSet 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.WriteLine("\nHere's " + username + "'s tweet(s):\n");

    rs = client.Query(null, stmt);
    while (rs.Next())
    {
        Record r = rs.Record;
        Console.WriteLine(r.GetValue("tweet"));
    }
}
else
{
    Console.WriteLine("ERROR: User record not found!");
}
}
finally
{
    if (rs != null)
    {
        // Close record set
        rs.Close();
```

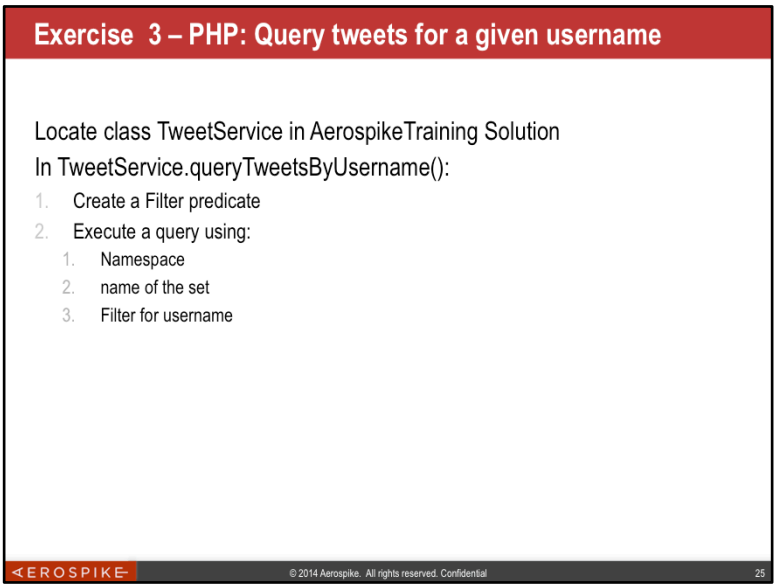


In TweetService.queryUsersByTweetCount(), locate these comments and add your code:

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" };



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("\nList of users with " + min
                        + "-" + max + " tweets:\n");

rs = client.Query(null, stmt);

while (rs.Next())
{
    Record r = rs.Record;
    Console.WriteLine(r.GetValue("username")
       + " has " + r.GetValue("tweetcount") + " tweets");
}
}
finally
{
    if (rs != null)
    {
     rs.Close();
```

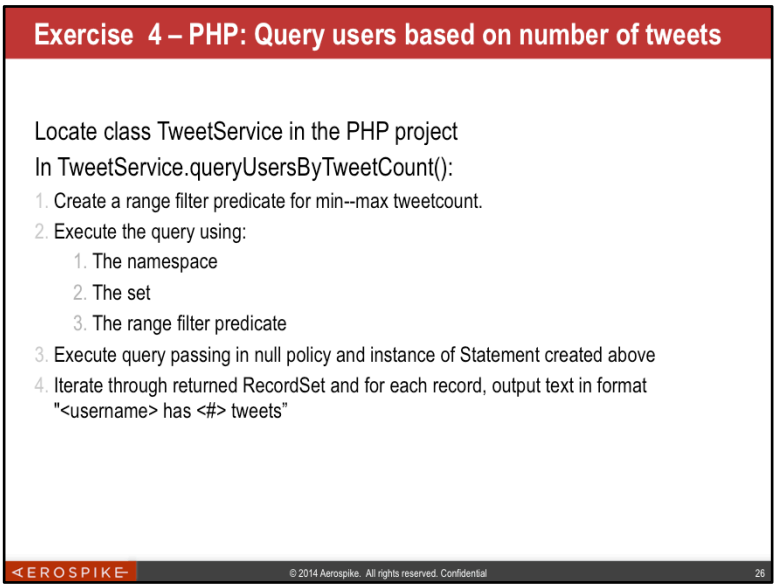

PHP Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
24



In TweetService.queryTweetsByUsername() add your 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

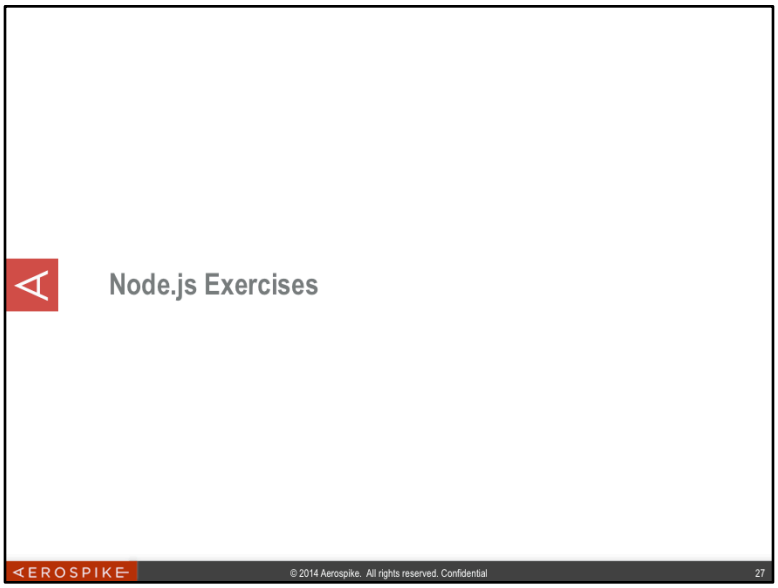
```
$where = $this->client->predicateEquals('username', $username);
$status = $this->client->query('test','tweets', $where, function
($record) {
    var_dump($record['bins']['tweet']);
}, array('tweet'));
if ($status !== Aerospike::OK) {
    // throwing an \Aerospike\Training\Exception
    throw new Exception($this->client, "Failed to query
test.tweets");
}
```

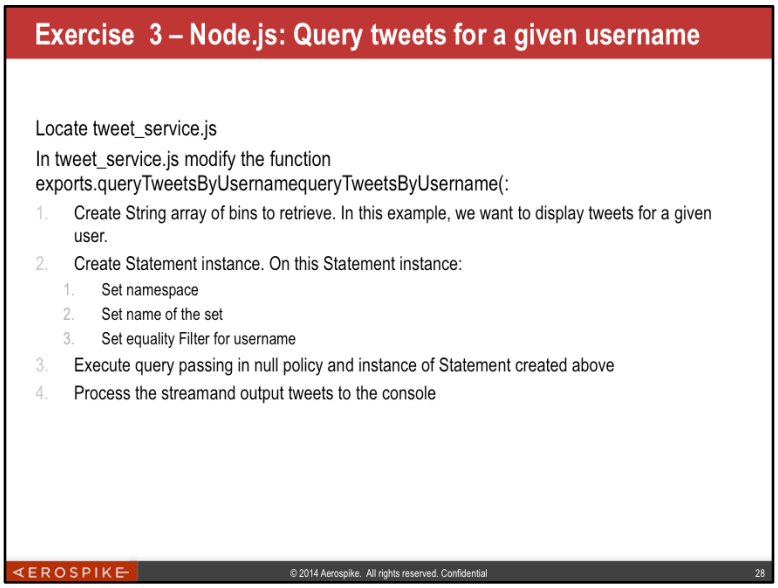


In TweetService.queryUsersByTweetCount(), add your code:

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"

```
$where = $this->client->predicateBetween('tweetcount', $min,
$max);
$status = $this->client->query('test','users', $where, function
($rec) {
    echo colorize("{$rec['bins']['username']} has
{$rec['bins']['tweetcount']} tweets", 'black')."\n";
});
if ($status !== Aerospike::OK) {
    // throwing an \Aerospike\Training\Exception
    throw new Exception($this->client, "Failed to query
test.users");
}
```


Node.js Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
27



In the function:
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

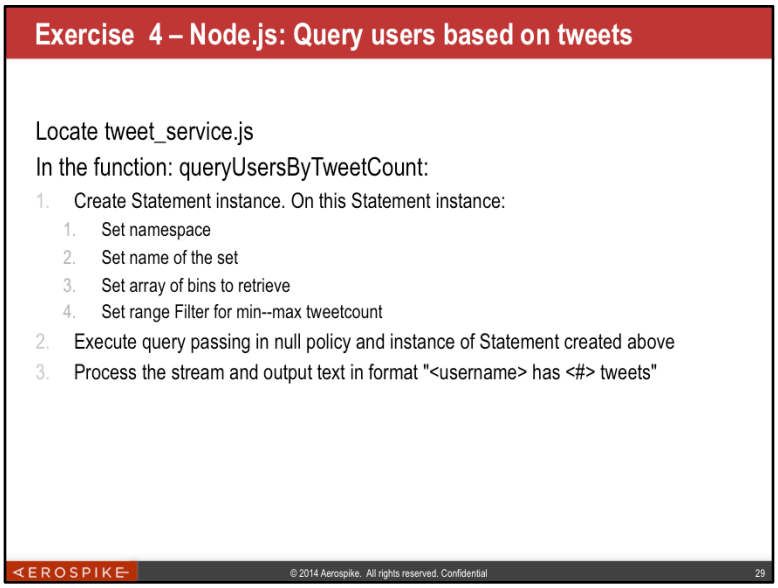
```
var statement = {filters:[aerospike.filter.equal('username',
answer.username)]};
```

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);
});
```



In the function 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

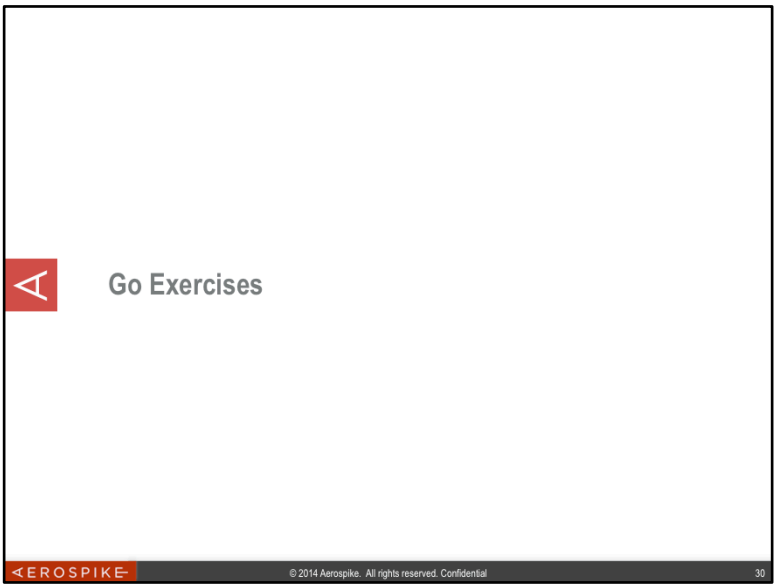
```
var statement = {filters:[aerospike.filter.range('tweetcount',
answers.min, answers.max)]};
statement.select = ['username', 'tweetcount'];
```

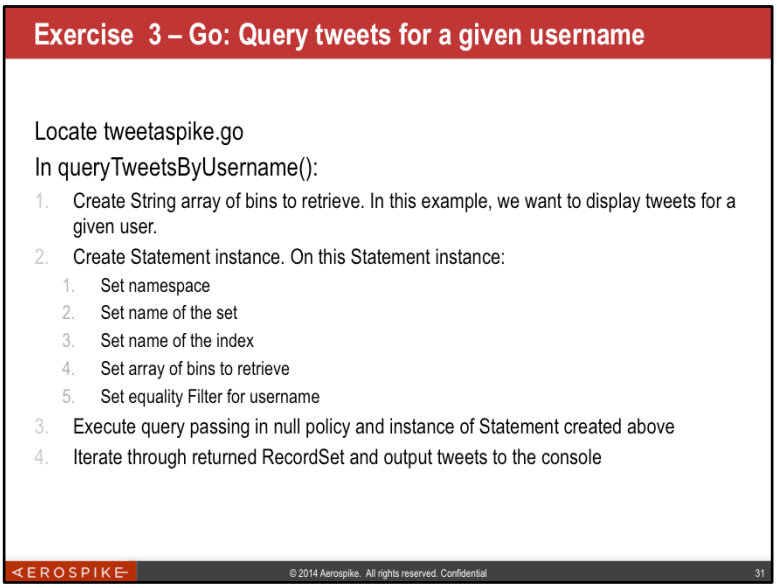
2. Execute query passing in null policy and instance of Statement created above

```
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!');
});
```

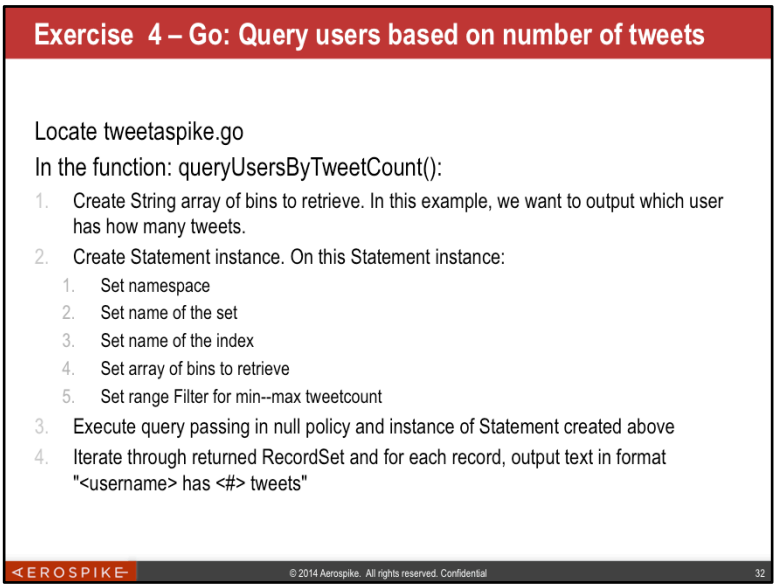

Go Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
30



In the function: queryTweetsByUsername(), locate these comments and 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
2. Execute the query from your code
3. Iterate through the RecordSet returned from the query
4. Close the record set

```
if len(username) > 0 {
     stmt := NewStatement("test", "tweets", "tweet")
     stmt.Addfilter(NewEqualFilter("username", username))

     fmt.Printf("\nHere's " + username + "'s tweet(s):\n")


     recordset, err := client.Query(nil, stmt)
     panicOnError(err)
L:
     for {
          select {
          case rec, chanOpen := <-recordset.Records:
               if !chanOpen {
                    break L
               }
               fmt.Println(rec.Bins["tweet"])
          case err := <-recordset.Errors:
               panicOnError(err)
          }
     }
     recordset.Close()

} else {
     fmt.Printf("ERROR: User record not found!\n")
}
```



In queryUsersByTweetCount(), locate these comments and 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

```
stmt := NewStatement("test", "users",
            "username", "tweetcount", "gender")
stmt.Addfilter(NewRangeFilter("tweetcount", min, max))
```

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

```
recordset, err := client.Query(nil, stmt)
panicOnError(err)
```

3. Iterate through returned RecordSet and for each record, output text in format "<username> has <#> tweets”

```
L:
for {
     select {
     case rec, chanOpen := <-recordset.Records:
          if !chanOpen {
               break L
          }
          fmt.Printf("%s has %d tweets\n", rec.Bins["username"],
                                              rec.Bins["tweetcount"])
     case err := <-recordset.Errors:
          panicOnError(err)
     }
}
```

4. Close the RecordSet

```
recordset.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


AEROSPIKE
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
34