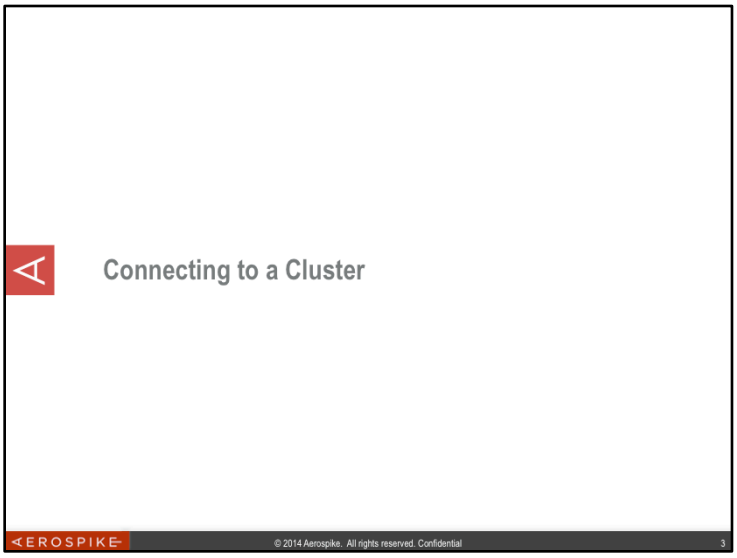
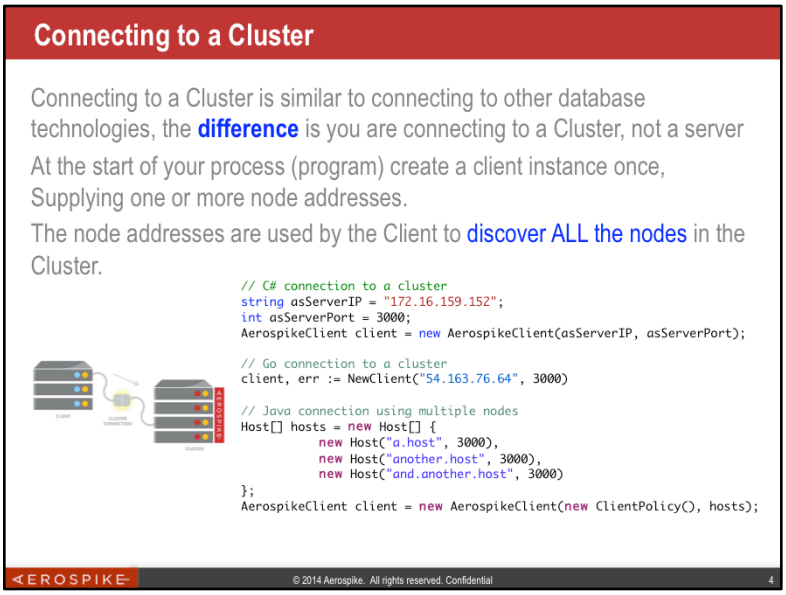
Key-value Operations

Goals

This module describes how to use Key-value operations. At the end of this module you will be able to code in C#, Java or Go to:

- Connect and Disconnect from an Aerospike cluster
- Perform Write operations
- Perform Read operations
- Apply advanced key-value techniques
- Correctly handle errors


Connecting to a Cluster
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
3



Connecting

Connecting to a Cluster is similar to connecting to other database technologies. You do not connect to a specific server. The first step is to create a AerospikeClient object specifying the IP address and port of one or more nodes in the cluster.

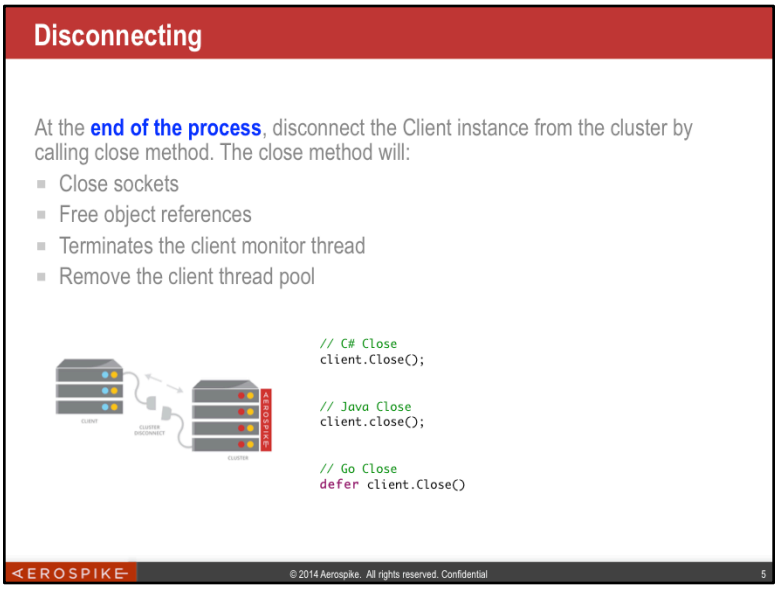
Initial nodes

You can specify a single node to connect to using the IP address and port when creating a new AerospikeClient object. The client to make initial contact to the specified node then discover all the nodes in the cluster. The best practice is to specify several nodes in the cluster when creating the client. The client will iterate through the array of nodes until it successfully connects to a node. Then through that node, the client will discover the other nodes in the cluster.

The client creates a maintenance thread which periodically pings nodes for the current state of the cluster.

Thread Safe

The AerospikeClient instance is thread-safe and can be used concurrently. Each get/set call is a blocking, synchronous network call to Aerospike. Connections are cached with a connection pool for each server node.



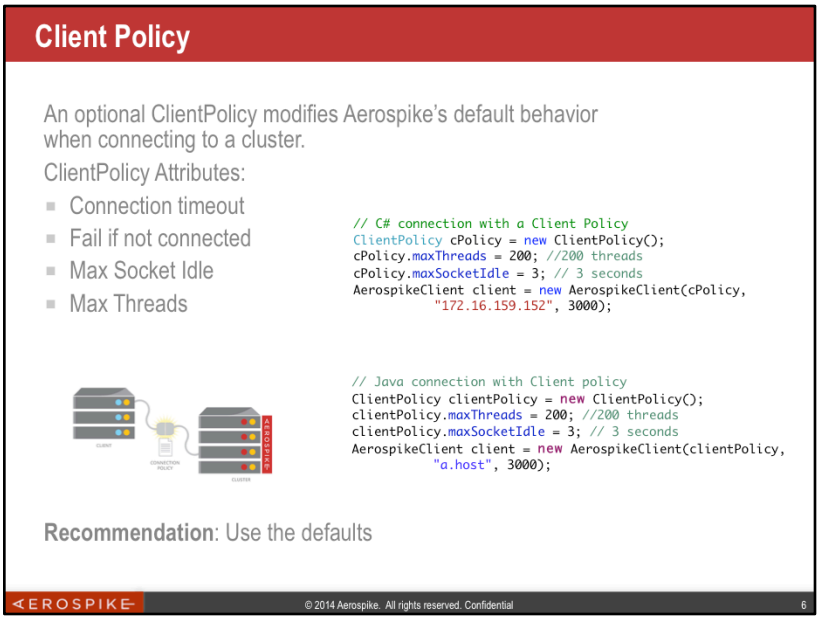
Cleanup

When all transactions are finished and the application is ready to have a clean shutdown, call the close method to remove the resources held by the client instance. The client instance is no longer usable once close has been called.

The close() function will:

- Close socket connections to every node in the cluster
- Free object references used by the Client instance
- Terminate the monitor thread
- Terminate threads in the thread pool and remove the pool.

Close should be called at the end of your process.



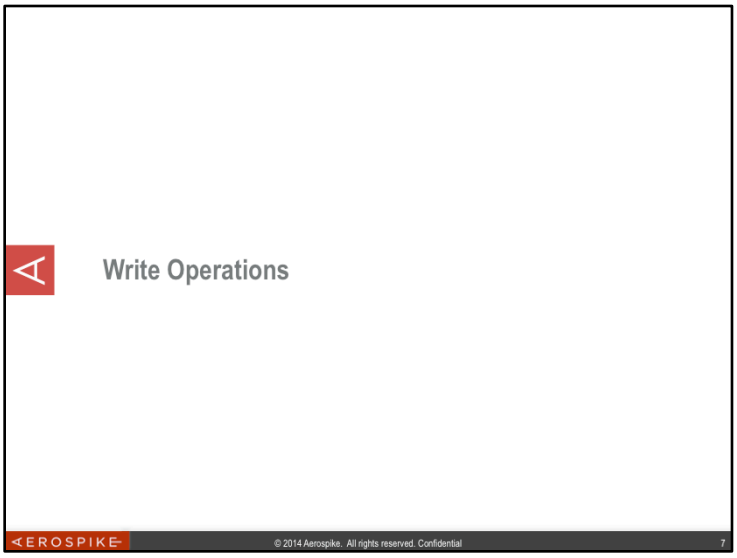
Client Policy

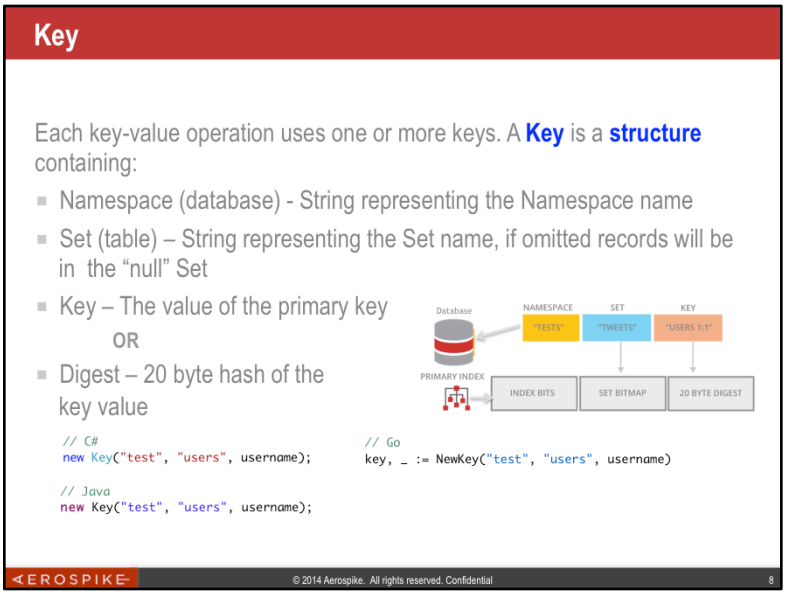
A Client Policy is container object/structure for attributes used in creating a connection to an Aerospike cluster. It modifies the default behavior when a Client instance connects to a cluster.

Recommendation

The default values for connecting to Aerospike have been established through extensive testing, and they will probably meet your needs. Start with the defaults and only change these attributes on consultation with Aerospike support.

Attribute	Type	Description
Timeout	Integer	Initial host connection timeout in milliseconds. The timeout when opening a connection to the server host for the first time.
Fail if not connected	Boolean	Throws an exception if host connection fails
Max Socket Idle	Integer	Maximum socket idle in seconds.
Max Threads	Integer	Maximum number concurrent threads using synchronous methods in the client instance.


Write Operations
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
7



Key

A Key is an object or structure that consists of a:

- Namespace
- Set – this is optional, if omitted the record is stored in the null Set
- The value of the primary key
- Digest

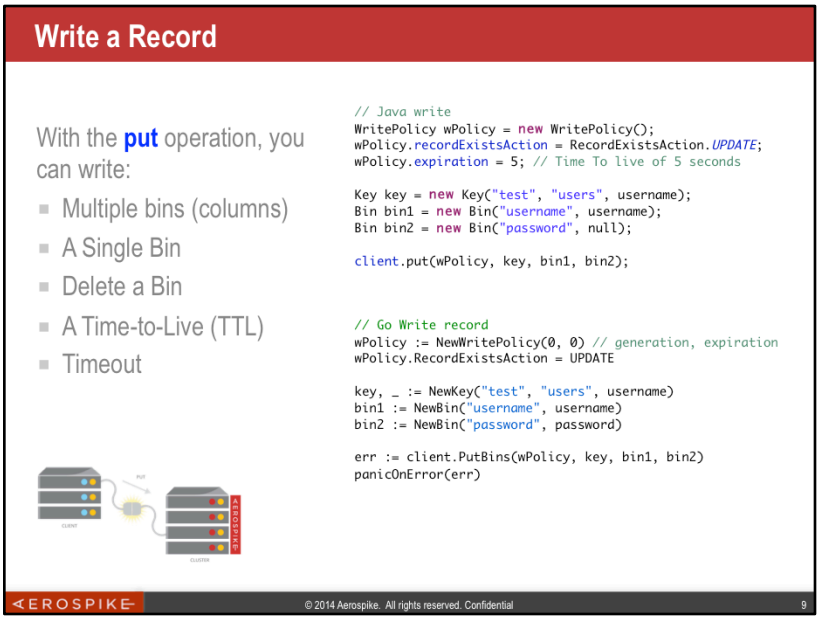
The Set name is mapped to an entry in a bit map in the primary index.

Digest

The primary key value is hashed with the RIPEMD160 algorithm to produce a 20 byte (160 bit) digest. RIPEMD160 always produces a 20 byte digest regardless of the size of key. By default, Aerospike does not store the key on the server.

Some operations (see scan, query, aggregate, etc.) return a Key object(or structure). This object does not, by default contain the original key, it contains the 20 byte digest.

You can use the digest in a Key object in place of the primary key value for most operations.



Write a Record

The put operation writes a record. You need to identify the record in the database via a key. A key can be a string, integer or blob.

The Set and Bins will be created automatically if they do not exist.

Write Values

One or more bins can be updated in the record by a Bin or an array of Bins to the put() call.

Delete Bin (column)

To delete a bin, simply set the bin value to be NULL:

Default behavior

- Create the record if the record doesn't exist.
- Update bin values in the record if the bins already exist.
- Add the bins if they do not exist.
- If the record already has bins, keep them in the record

WritePolicy

The policy specifies the transaction's re-transmission policy, timeout interval, record expiration, and what to do if the record already exists. Instantiating the policy initializes client defaults

To change the write behavior through the write policy object. For example: Only write if the record does not already exist or completely replace a record, ONLY if it exists

Write Record with Time-to-live (TTL)

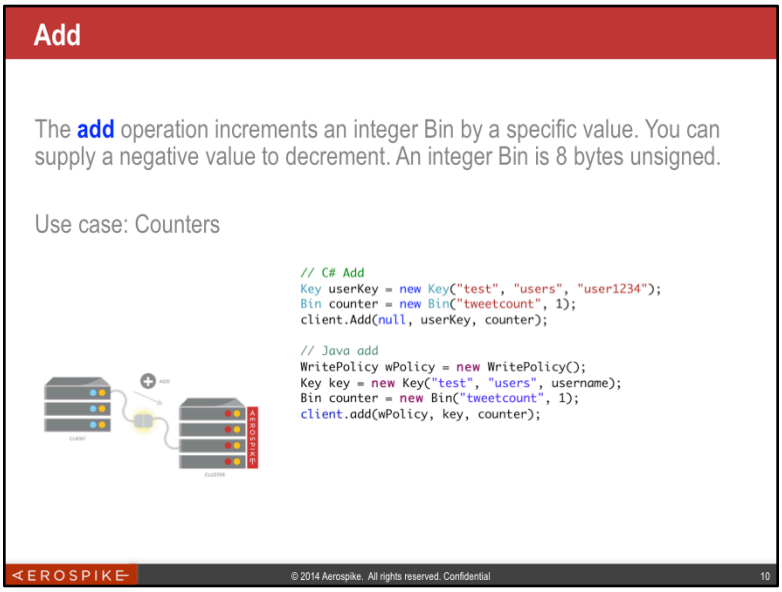
It is possible to specify a time-to-live value for a record on a write. The expiration units are seconds.

writePolicy.expiration = 2; // 2 seconds

If a 0 is specified, then the default time-to-live specified on the server side will be applied each time a record is updated. If a -1 is specified, then the record will never be expired.

Replica Writes

Write to replicas are controlled in the Namespace configuration and not on a per operation basis.



Add

The Add operation increments, or decrements an integer Bin value(only) by the amount specified. An integer Bin is 8 bytes unsigned.

To decrement, you supply a negative value.

One or more Bin parameters are passed to the operation and all increments/decrements are atomic in the operation on the specified record.

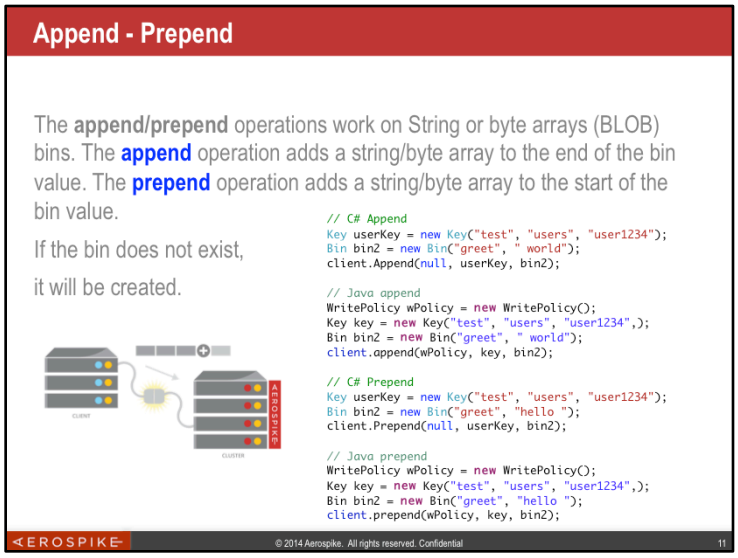
The write policy specifies the transaction timeout, record expiration and how the transaction is handled when the record already exists.

Counters

Many applications require a counter, these are simple to implement in Aerospike.

In the section “Techniques” we cover:

- How to avoid frequent updates and hot keys
- Counter increment and read (Atomic)



Append

The **append** operation adds a string to the end of a string bin value, the **prepend** operation adds the value to the start of the value.

One or more Bin parameters are passed to the operation and the all appends/prepends are atomic in the operation on the specified record.

The **write policy** specifies the transaction timeout, record expiration and how the transaction is handled when the record already exists.

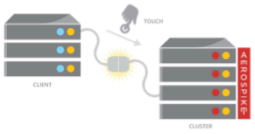
If the bin does not exists, it will be created.

Note: If you append a String to an Integer Bin you will get an error.

Touch

The **touch** operation resets one, *or more*, records' time to expiration default configured in the namespace. The defaults can be changed using a WritePolicy.

A multi–record operation is **not** atomic.



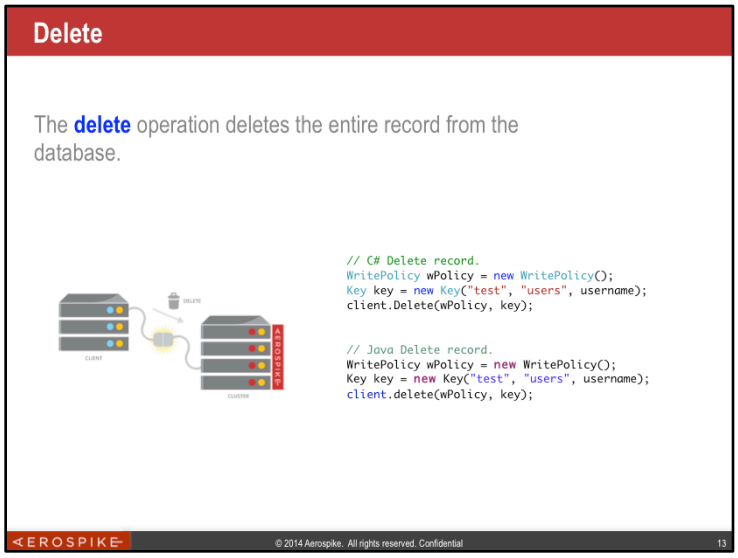

```
// C# Touch
Key userKey = new Key("test", "users", "user1234");
Key userKey2 = new Key("test", "users", "user1235");
client.Touch(null, userKey, userKey2);

// Java touch
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.touch(wPolicy, key);
```

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 12

Touch

The **touch** operation resets one, *or more*, records' time to expiration to the policy's expiration or the Namespace default if a policy is omitted.

Providing more than one Key will allow you to "touch" many records in one operation. Note: Multi–record touch operation is **not** atomic.

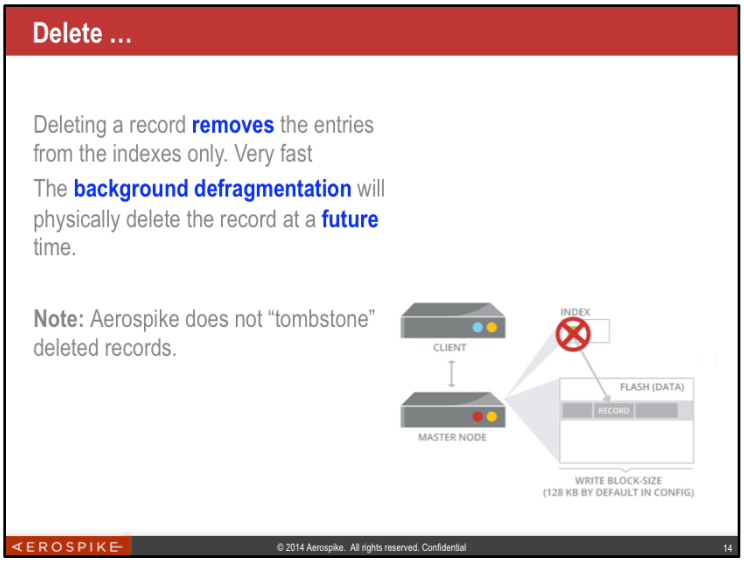


Delete

When a record is deleted, the index entries in the primary index, and any secondary indexes, are removed.

At some future time, the automatic defragment process (thread) will remove the record from the storage layer when it defragments the storage block where the record is located.

This has the unusual side effect that a deleted record can be “resurrected” on a server restart if the server is configured to load data when it starts.



Deleting a record in Flash

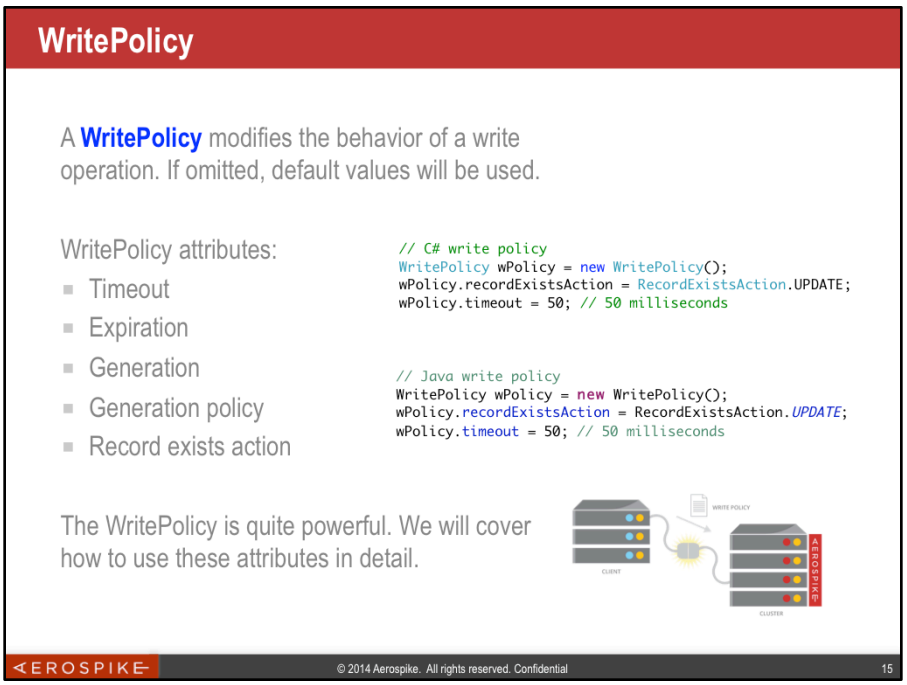
Deleting a record **removes** the entries from the indexes only. Very fast

The background defragmentation will physically delete the record at a **future** time.

The process:

1. Client finds Master Node from partition map.
2. Client makes delete request to Master Node.
3. Master node removes the entry
4. from index(es)
5. Master Node coordinates deletion with replica nodes (not shown).
6. Master Node returns success to client.
7. Data is eventually deleted from flash by defragmentation.

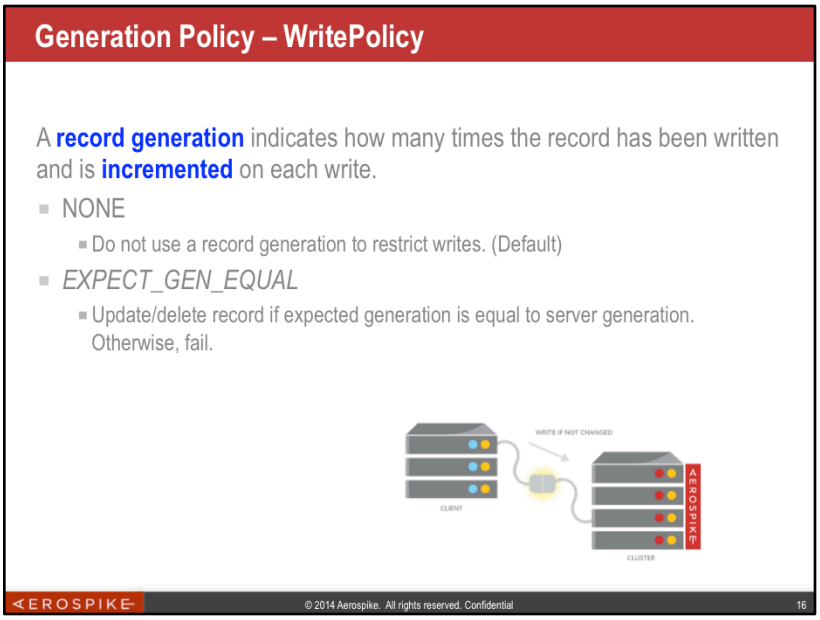
Note: Aerospike does not tombstone deleted records. If a server is restarted AND data is reloaded from storage, there is potential for "deleted" records to be "resurrected". This only occurs if the defragmentation task has not processed a storage block containing the deleted record.



WritePolicy

WritePolicy extends Policy and therefore has the same attributes, plus it has attributes specific for Write operations. A WritePolicy is optional and, if omitted, default values will be used. You can use a

Attribute	Type	Description
Timeout	Integer	In milliseconds. Default is no timeout (0) - This actually means the server default timeout.
Expiration	Integer	Record expiration is also known as time to live (TTL), It is the number of seconds a record will live before being removed by the server. Expiration values: • -1: Never expire. • 0: Default to namespace configuration variable "default-ttl" on the server. • > 0: Actual expiration in seconds.
Generation	Integer	Expected generation. Generation is the number of times a record has been modified (including creation) on the server. If a write operation is creating a record, the expected generation would be 0.
Generation policy	Enumeration	Qualify how to handle record writes based on record generation. The default (NONE) indicates that the generation is not used to restrict writes. Values: NONE, EXPECT_GEN_EQUAL, EXPECT_GEN_GT, DUPLICATE
Record exists action	Enumeration	Qualify how to handle writes where the record already exists. Values: CREATE_ONLY, REPLACE, REPLACE_ONLY, UPDATE_ONLY



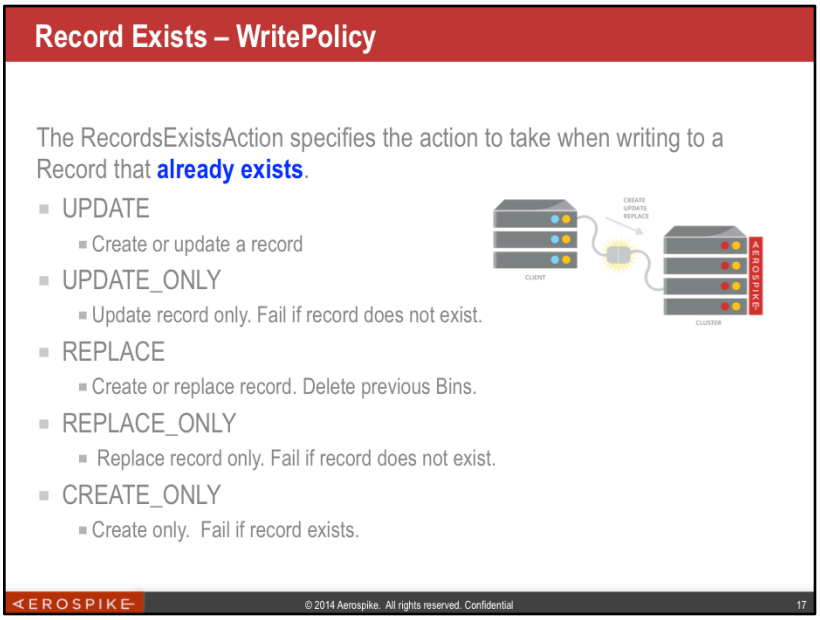
Generation Policy

Aerospike uses multi-version concurrency control. The record generation indicates how many times the record has been written and is incremented on each write.

The generation policy is a field in the write policy. Valid values are:

- NONE - Do not use record generation to restrict writes. (DEFAULT)
- *EXPECT_GEN_EQUAL* - Update/delete record if expected generation is equal to server generation. Otherwise, fail.

A generation policy is commonly used in a read-modify-write pattern where the generation policy is set to EXPECT_GEN_EQUAL. The write operation will expect the record generation to be equal to the value read previously, this would indicate that the record has not been written by another operation.



Record Exists

The RecordsExistsAction specifies the action to take when writing to a Record that already exists.

The valid policy values are:

- Update: Create or update a Record

Value	Description
UPDATE	Create or update record. Merge write command bins with existing bins.
UPDATE_ONLY	Update record only. Fail if the record does not exist. Merge write command bins with existing bins. This is like SQL UPDATE …
REPLACE	Create or replace record. Delete existing bins not referenced by write command bins.
REPLACE_ONLY	Replace record only. Fail if record does not exist. Delete existing bins not referenced by write command bins.
CREATE_ONLY	Create only and fail if record exists. This is like SQL INSERT …

Note:

Replacing a whole record has significantly better ***storage*** performance than replacing a subset of Bins in a record.

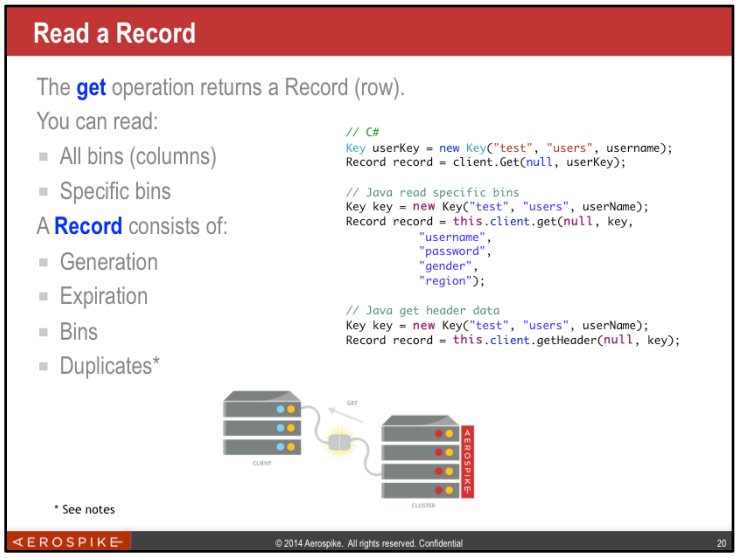
When you know that you are replacing, please use the REPLACE flag.

Deleting a Bin

To delete one or more Bins in an operation, set the Bin value to null.

Note: Deleting Bins has a similar storage cost of writing Bins in a Record.


Read Operation
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
19



Read a Record

The read operation allows you to:

- Read all Bins in a Record. Remember – a Bin is like a column.
- Read specified Bins in a Record.
- Read only the Generation and Expiration of a record.

Each of the methods returns a Record, which contains Expiration, Generation, and Bins of the record.

Read All Bins of a Record

To read all the Bins of a Record, call the get() method without a list of Bins. This will read the entire Record, including Generation and Expiration from the database, and return a Record Object.

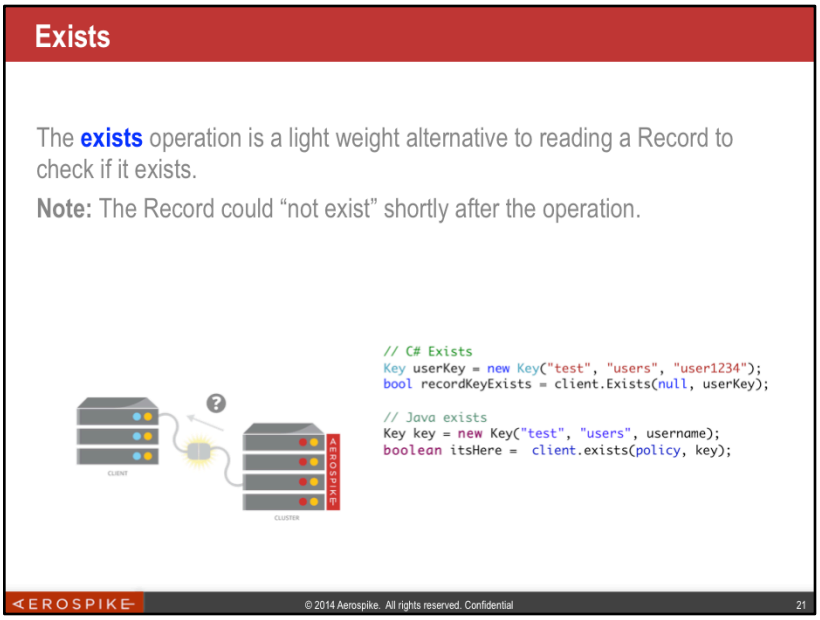
Read Specific Bins of a Record

To read specific bins of a Record, call the get() method and specify the names of the Bins to be returned.

Record

A Record consists of:

- Expiration - the date record will expire, in seconds from Jan 01 2010 00:00:00 GMT
- Generation – a count of the modifications to the record.
- Bins – A map/dictionary of requested name/value bins.
- *Duplicates - List of all duplicate Records (if any) for a given key. Duplicates are only created when the server configuration option "allow-versions" is true (default is false) and client RecordExistsAction.DUPLICATE policy flag is set and there is a Generation error. Almost always null.



Exists

This operation is a light weight alternative to reading the Record. It checks the existence of a Record and returns a Boolean.

The Exists operation does not hold a lock, it tests existence at the time you call it. Checking to see if a Record exists and then reading it (if true) is not a sensible approach. Why? Because another client instance could have successfully deleted the Record between your **exists** and **read** operations.

Batch reads

The **batch** operation allows you to read many Records in one operation

- Array of Keys
- Records returned in the same order as the Keys


```
// C# batch
// Create an array of keys
Key[] keys = new Key[tweetCount];
for (int i = 0; i < keys.Length; i++)
{
      keys[i] = new Key("test", "tweets",
            (username + ":" + (i + 1)));
}
// Initiate batch read operation
Record[] records = client.Get(null, keys);
```

```
// Java batch
// Create an array of keys
Key[] keys = new Key[27];
for (int i = 0; i < keys.length; i++) {
      keys[i] = new Key("test", "tweets",
      (username + ":" + (i + 1)));
}
// Initiate batch read operation
Record[] records = client.get(null, keys);
```

AEROSPIKE 22

Batch Reads

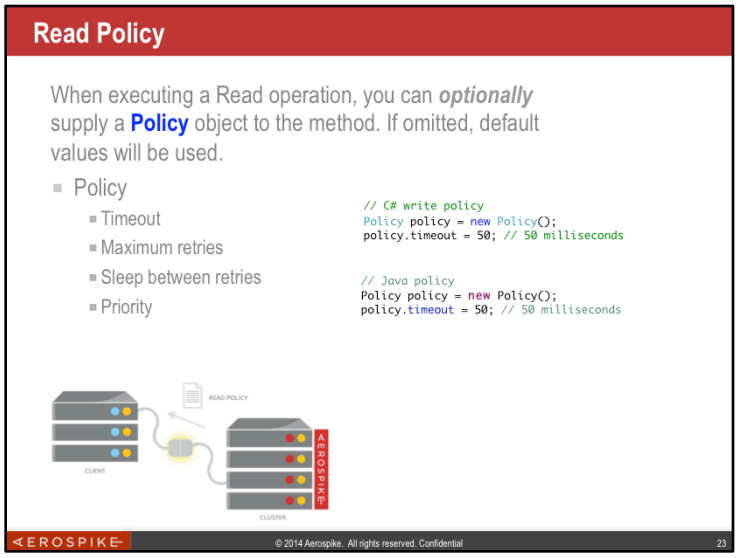
In addition to reading single Record each time, it is also possible to read multiple Records from the cluster in one operation.

Internal to the client, the operation will group the keys based on which Aerospike Server node is best to handle the request, and uses a thread pool to handle the requests to all nodes concurrently. All network requests are combined efficiently and sockets are reused.

After all the nodes have returned the Record data, the Records will be return to the caller.

The array of Records returned is in the same order as the Keys passed in. If a Record is not found on the database, the array entry will be NULL.

This is not an atomic operation, Records are read as they are found.

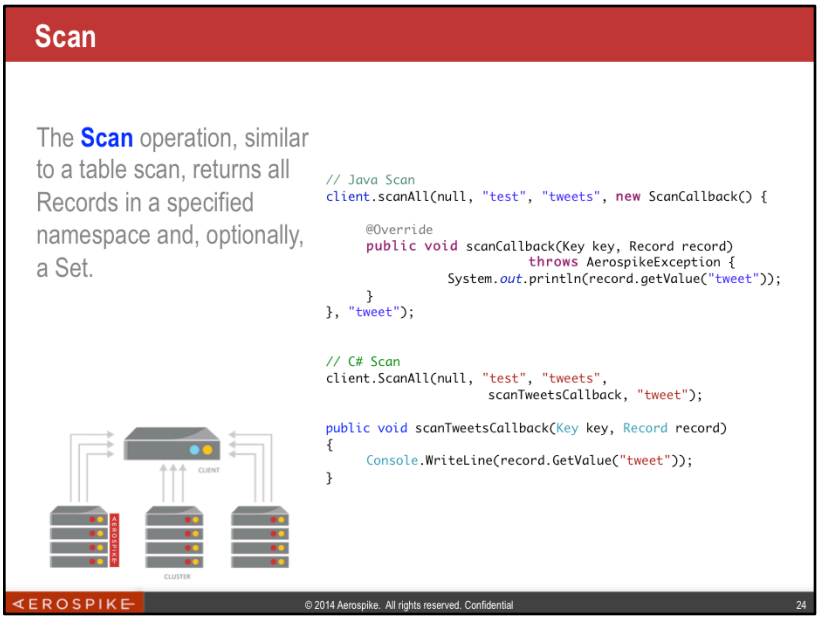


Policies

Policies are container objects/structures for transaction policy attributes used in all database operation calls. A Read Policy object can be optionally on each Read operation, to modify default values.

Attribute	Type	Description
Timeout	Integer	Transaction timeout in milliseconds
Max retries	Integer	Maximum number of retries before aborting the current transaction.
Sleep between retries	Integer	Milliseconds to sleep between retries if a transaction fails and the timeout was not exceeded.
Priority	Constant	Priority of request relative to other transactions. DEFAULT, LOW, MEDIUM, HIGH

NOTE: Timeout trumps retries. If your timeout is 10ms, number of retries is 3 and sleep between retries is 5ms, you will timeout before the 3rd retry.



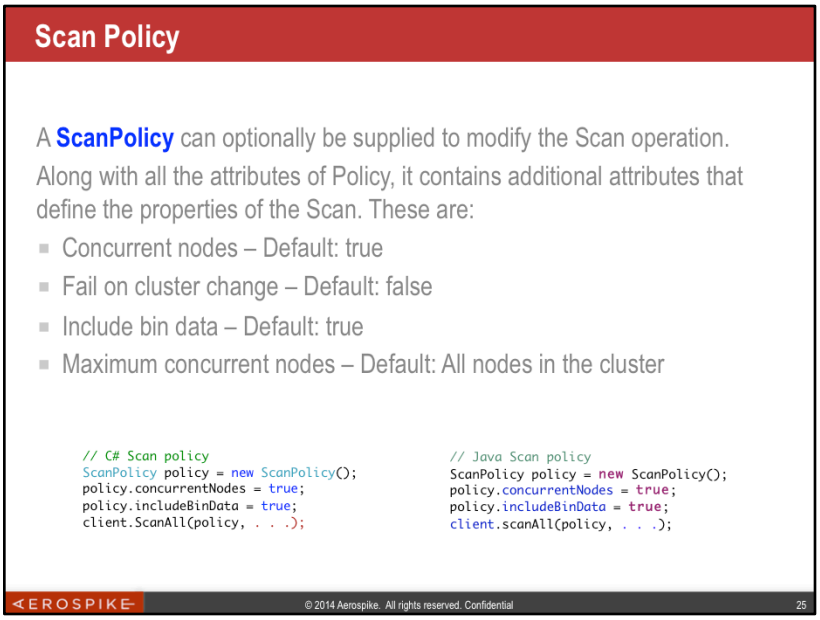
Scan Records

The Scan operation provides the ability to read all Records in a specified Namespace and, optionally, a Set.

First, the Scan Policy is initialized. in the Policy Object, If **concurrent nodes** = **true**, the Scan will query nodes for Records in parallel using threads. Otherwise, Scan will query each node one at a time in sequence. If **include bin data** is set in the policy object, specified Bins (all bins by default) will be returned with each Record.

If an exception is thrown, parallel Scan threads to other nodes will also be terminated and the exception will be propagated back through the initiating Scan call.

Scan returns Records to the user defined callback. Multiple threads will likely be calling your Scan callback in parallel. Therefore, your Scan callback implementation should be thread safe.



Scan Policy

Used to control the optional parameters in a scan operation

- Concurrent nodes – Issue scan requests in parallel, the default is true
- Fail on cluster change – Terminate scan if cluster is adding or removing a node, the default is false
- Include bin data - Indicates if bin data is retrieved. If false, only record digests are retrieved. The default is true.
- Maximum concurrent node – The maximum number of concurrent requests to server nodes, the default is all nodes in the cluster. This is only valid when **Concurrent nodes** is true.


Key-value Techniques
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
26

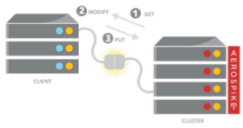
Read Modify Write (CAS)

A common usage pattern is to do Read-Modify-Write of a record (or **Check-and-Set**).

Example: modify a JSON document in the client

This involves:

- Reading the Record.
- Modifying the Record at the application.
- Set the WritePolicy to be **EXPECT_GEN_EQUAL**.
- Write the modified data with the generation previous Read.



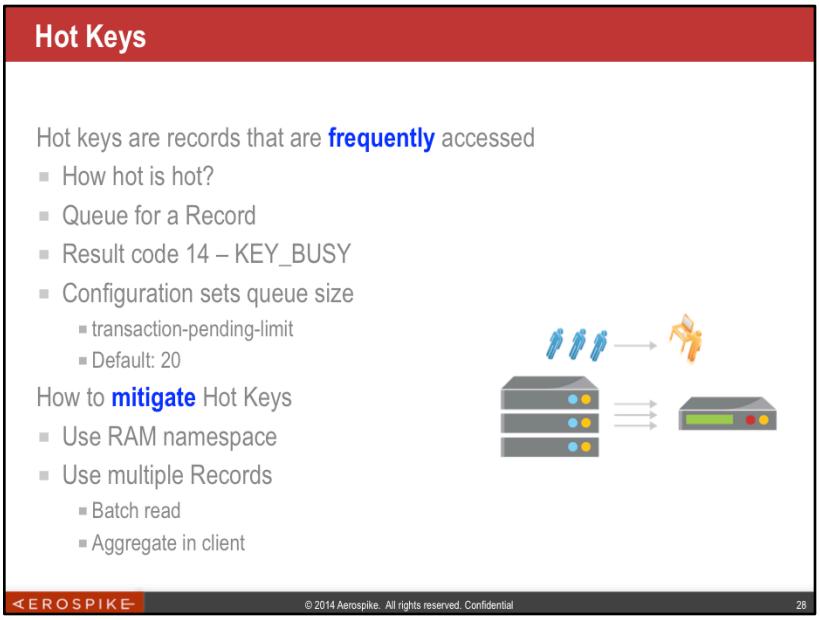
```
// Java read-modify-write
WritePolicy wPolicy = new WritePolicy();
wPolicy.generationPolicy = GenerationPolicy.EXPECT_GEN_EQUAL;

Key key = new Key("test", "users", username);
Bin bin1 = new Bin("username", username);
Bin bin2 = new Bin("password", password);

client.put(wPolicy, key, bin1, bin2);
```


Read-Modify-Write

A common usage pattern is to do Read-Modify-Write of a record (or Check-and-Set). This involves:

- Reading the Record.
- Modifying the Record in the application.
- Set the WritePolicy to be EXPECT_GEN_EQUAL.
- Write the modified data with the generation previous Read.

If the generation number of the record in the cluster the same, the Write will proceed. If the generation number is different, the Record has been written by another request and the Write will fail. The data in your application is now stale and you will need to re-read the Record.



Hot Keys

A hot key is a Record that is frequently accessed by a large number of (almost) concurrent requests. This is often occurs with a counter that counts every action, request, click, etc.

How hot is hot? – Depending on the hardware you can have a single key with over 100K tps, and depending on the operation.

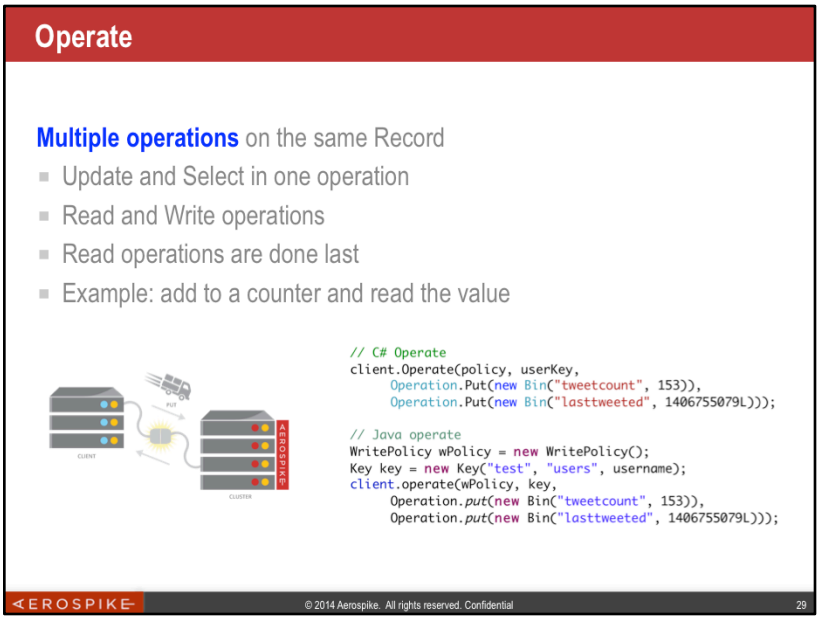
If the number of requests exceeds the queue size for a key, the client well receive the result code 14 indicating that the key is busy.

The size of the queue is controlled by the configuration setting transaction-pending-limit, located in the **Service** stanza of the Aerospike configuration file. The default value is 20. This means that 20 concurrent requests will be accepted before the client would be returned **KEY_BUSY**

How to Mitigate Hot Keys

Put high Hot Keys in a RAM namespace.

Separate high frequency hot key counters in to separate records e.g. - Instead of a single counter for every click on a URL, have N counters segmented by a demographic (location, age group, favorite color, etc.) and stored in separate Keys. Aggregate the value in your application by using a Batch read.



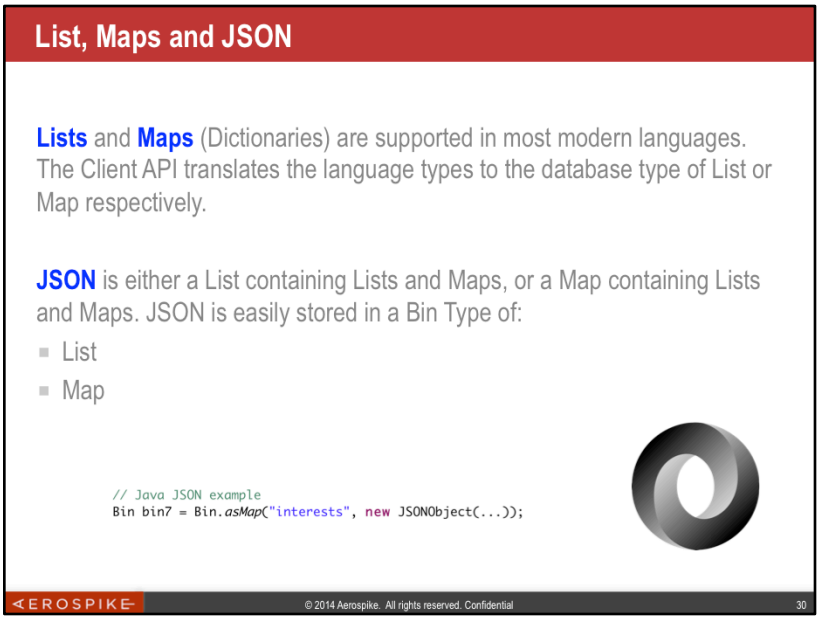
Multiple Operations on a Record

The Client API provides a way to perform multiple operations on a record within a single transaction. This feature also allows the bins of a Record to be modified and read back to the client within a single transaction, i.e. it allows an application to perform an atomic modification that returns the result.

It is like an **Update** and a **Select** in one atomic operation. You package a set of operations into the request, the 'write operations are done first, then the read operations. The record is returned to the client at the end of the operation.

The following are the operations which can be performed using the C# or Java client API.

Operation	Description	Conditions
Write	Write a value to a bin.	
Read	Read the value of a Bin	
read_header	Read only the meta-data (generation and time-to-live) of the record.	
Add	Add an integer to the existing value of a bin.	Existing value must be integer.
Append	Append string to the existing value of a bin.	Existing value must be string.
Prepend	Prepend string to the existing value of a bin.	Existing value must be string.
Touch	Rewrite the same record	Generation and Time-to-live will be updated.



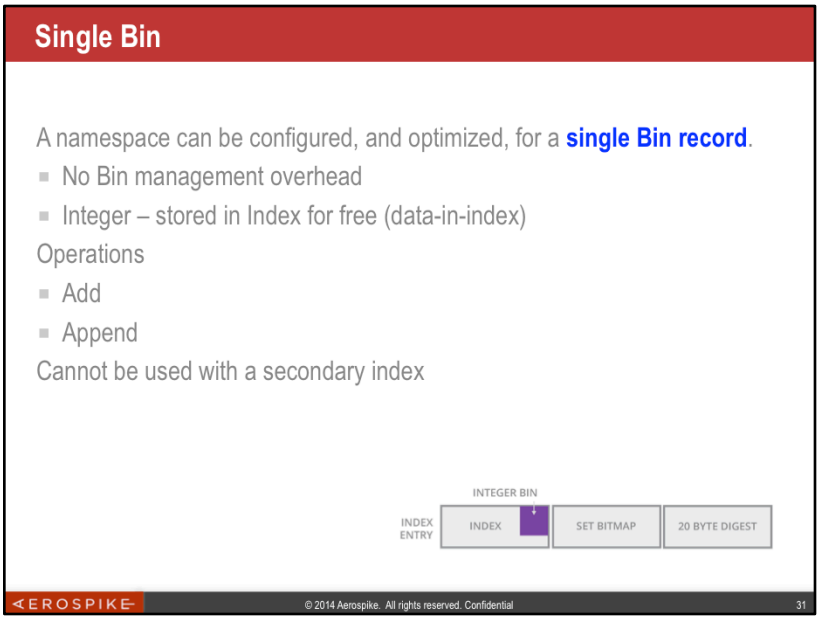
Lists, Maps and JSON

Aerospike supports the database native times of List and Map. Lists and Maps can contain other Lists and Maps, so you can store complex objects in a single Bin and read or write the whole object from the client API.

Because the List or Map is stored in a database native format, you can, for example, write it in C# and read it in Python. Also, the database native type can be manipulated by User Defined Functions (UDFs).

JSON

A JSON object is either a Map(Dictionary) containing Lists and Maps, or a List containing Lists and Maps. This means that a JSON object can easily be stored in either a List or a Map in a database native way.



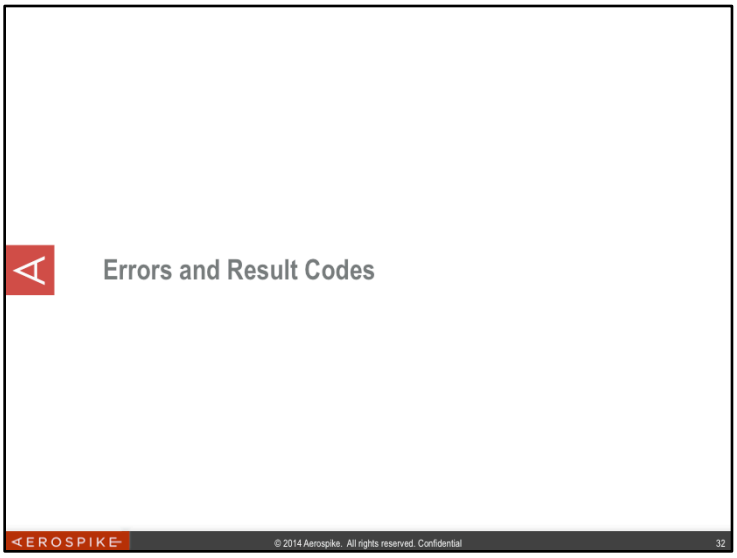
Single Bin

A namespace can be configured for a single Bin record as apposed to a multi-Bin record. The simplifies the storage and record management parts of Aerospike and can provide some performance gains. There is no processing or storage overhead for Bin names and values.

Single bin namespaces (like all namespaces) need a cluster wide restart, after they are defined, to take effect – so PLAN AHEAD!

Integer single Bin namespaces

You can define a single bin namespace in RAM where the integer Bin can be stored in the Primary index. This is not only very fast, but there is no other RAM requirement other than the index entry it's self.


Errors and Result Codes
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
32

Error Handling

If an error occurs as a result of the database operation, an AerospikeException is thrown.

An AerospikeException has a ResultCode field, which describes the exact cause of the error given by the Aerospike Server; and human readable message describing the error.

There can also be conditions where an error occur on the client side, and a sub-classed AerospikeException is thrown. Examples of such conditions include:

- AerospikeException.Connection - Exception thrown when client can't connect to the server.
- AerospikeException.Timeout - Exception thrown when database request expires before completing.

General result codes

These are the most common result/error codes you will see as a developer:

- KEY_NOT_FOUND_ERROR
- GENERATION_ERROR
- KEY_EXISTS_ERROR
- BIN_EXISTS
- TIMEOUT
- BIN_TYPE_ERROR
- KEY_BUSY
- BIN_NOT_FOUND
- KEY_MISMATCH

A full list is included in the notes

AEROSPIKE 34

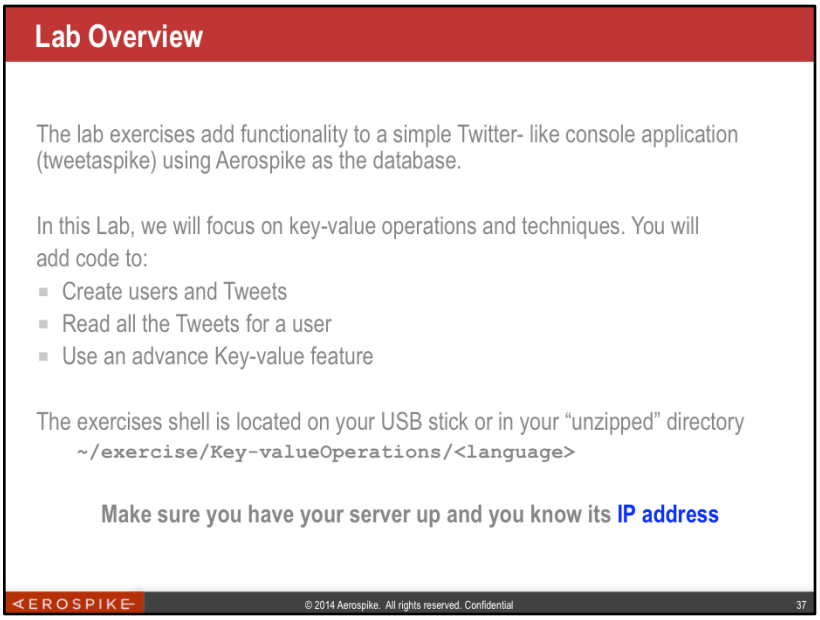
Code	Symbolic name	Description
0	OK	Operation was successful.
1	SERVER_ERROR	Unknown error in the server. Contact Aerospike support.
2	KEY_NOT_FOUND	On retrieving, touching or replacing a record that doesn't exist.
3	GENERATION_ERROR	On modifying a record with unexpected generation.
4	PARAMETER_ERROR	Bad parameter(s) were passed in database operation call.
5	KEY_EXISTS_ERROR	On create-only (write unique) operations on a record that already exists.
6	BIN_EXISTS_ERROR	On create-only (write unique) operations on a bin that already exists.
7	CLUSTER_KEY_MISMATCH	Expected cluster ID was not received – Cluster is changing during a Scan
8	SERVER_MEM_ERROR	Server has run out of memory.
9	TIMEOUT	Client or server has timed out.
10	NO_XDS	XDR product is not available.
11	SERVER_NOT_AVAILABLE	Server is not accepting requests.
12	BIN_TYPE_ERROR	Operation is not supported with configured bin type (single-bin or multi-bin).
13	RECORD_TOO_BIG	Record size exceeds limit.
14	KEY_BUSY	Too many concurrent operations on the same record.
15	SCAN_ABORT	Scan aborted by server.
16	UNSUPPORTED_FEATURE	Unsupported Server Feature (e.g. Scan + UDF).
17	BIN_NOT_FOUND	Specified bin name does not exist in record.
18	DEVICE_OVERLOAD	The server node's storage device(s) can't keep up with the write load.
19	KEY_MISMATCH	Key type mismatch. – Digest collision – never been seen
100	UDF_BAD_REPONSE	A user defined function returned an error code.


Lab: Key-value Operations
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
35

Objective

After successful completion of this Lab module you will have:

- Connected to a cluster
- Written and read records using simple and complex values
- Used advanced key-value techniques



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

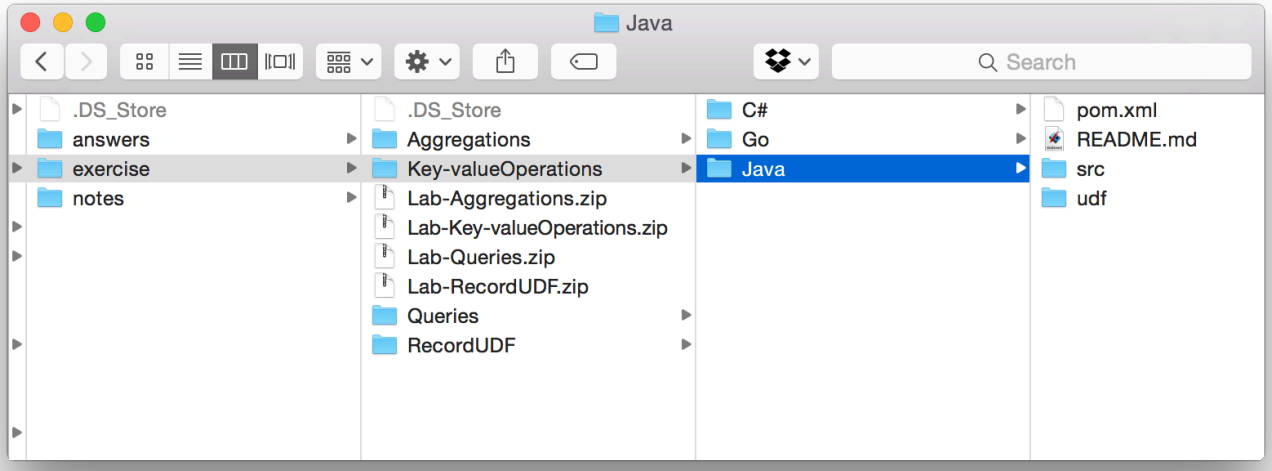
- Answers
- Exercise
- Notes

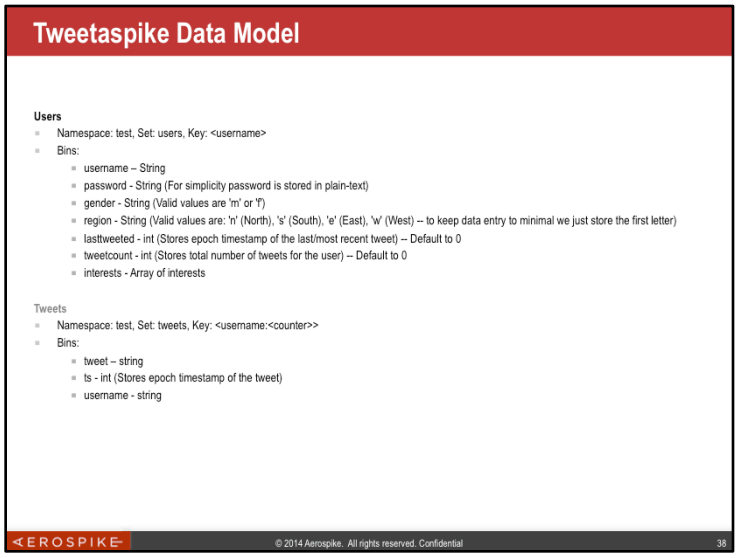
In the exercise directory, select the subdirectory for your programming language:

- C#
- Java
- Go

The exercises for this module are in the 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>

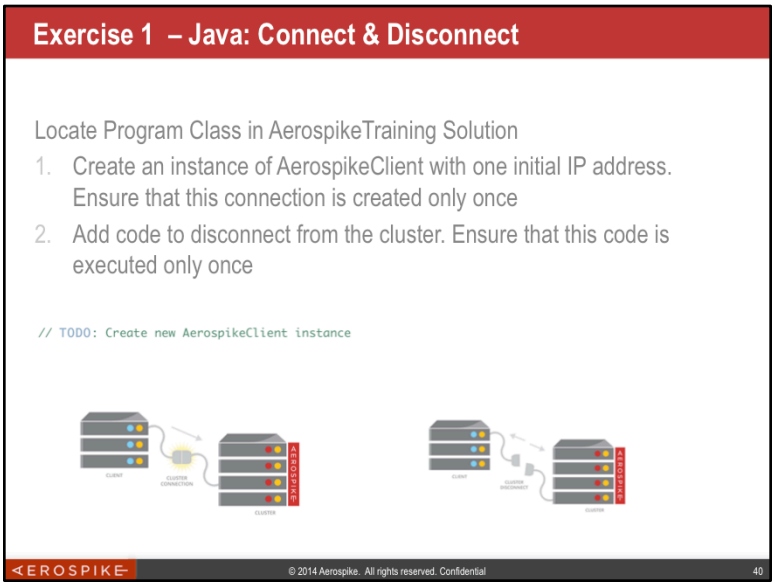
Bin name	Type	Comment
username	String	
password	String	For simplicity password is stored in plain-text
region	String	Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) -- to keep data entry to minimal we just store the first letter
lasttweeted	Integer	Stores epoch timestamp of the last/most 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
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
39



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();
}
```

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 = Bin.asList("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. Create a WritePolicy

2. Create Key and Bin instances

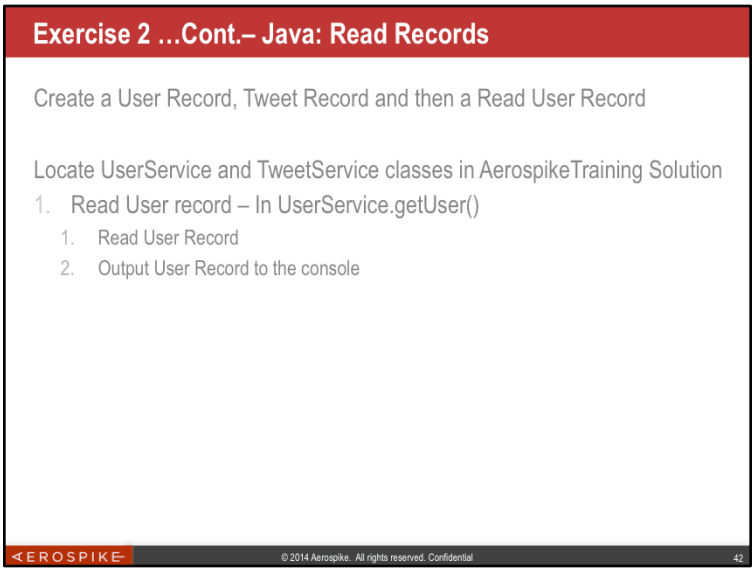
```
tweetKey = new Key("test", "tweets", username + ":"
                                     + nextTweetCount);
Bin bin1 = new Bin("tweet", tweet);
Bin bin2 = new Bin("ts", ts);
Bin bin3 = new Bin("username", username);
```

3. Write a tweet record using the Key and Bins

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

4. Update the user record with tweet count

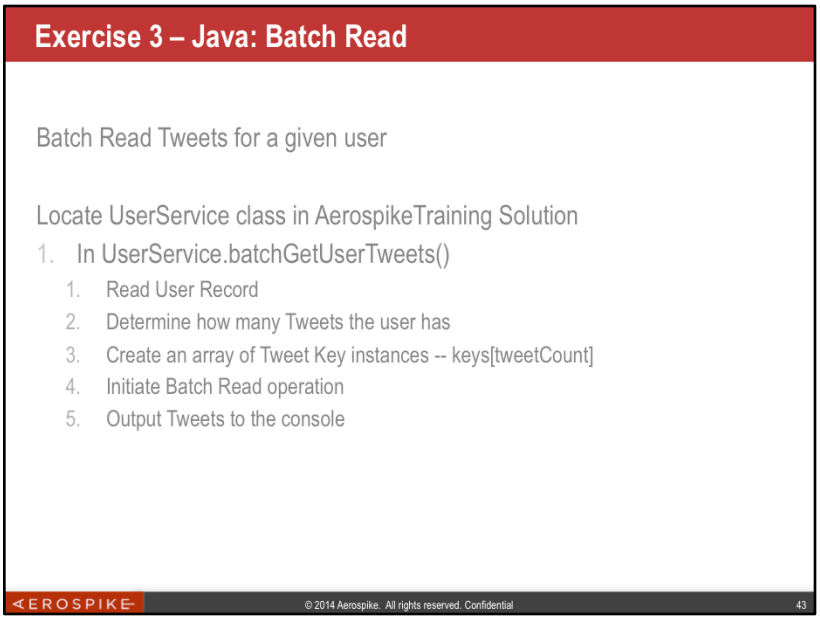
```
// Update tweet count and last tweet'd timestamp in the user
// record
long ts = getTimeStamp();
updateUser(client, userKey, wPolicy, ts, nextTweetCount);
```



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");
}
```



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 = (Integer) userRecord.getValue("tweetcount");
```

3. Create a "list" of tweet keys

```
// Create an array of keys so we can initiate batch read
// operation
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
5. Then print out the tweets

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

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

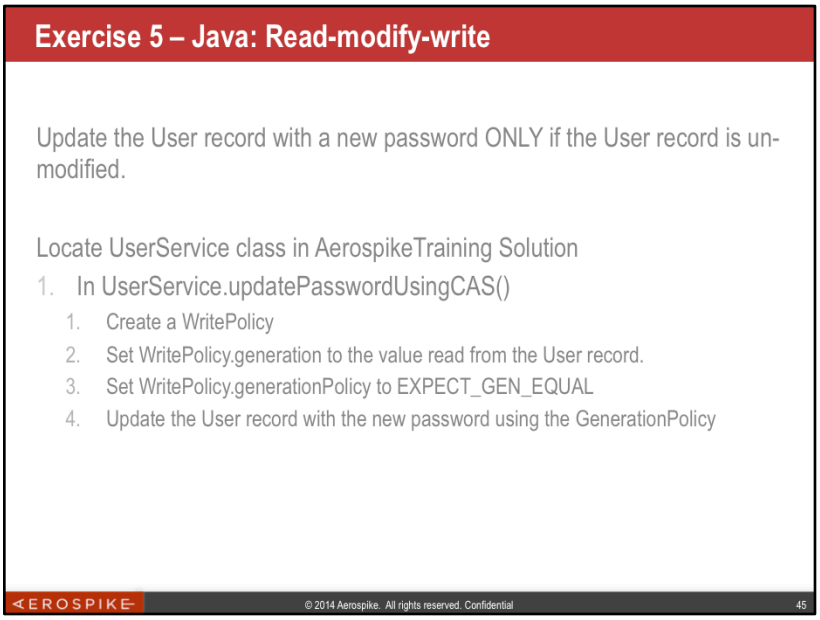
In the TweetService.scanAllTweetsForAllUsers(), add code similar to this:

1. Create a ScanPolicy
2. Set policy parameters

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

3. Perform a Scan operation and process the results

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

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

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

     }
}, "tweet");
```



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

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);
}
```

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

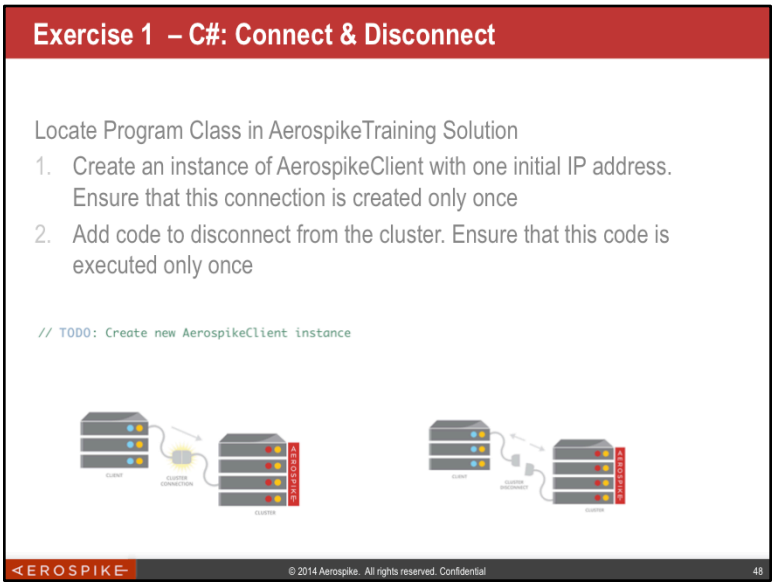
2. Uncomment the line:

```
// TODO: Update tweet count and last tweeted timestamp in the user record using Operate
// Exercise 6
// updateUserUsingOperate(client, userKey, policy, ts);
```

3. 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());
```


C# Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
47



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.

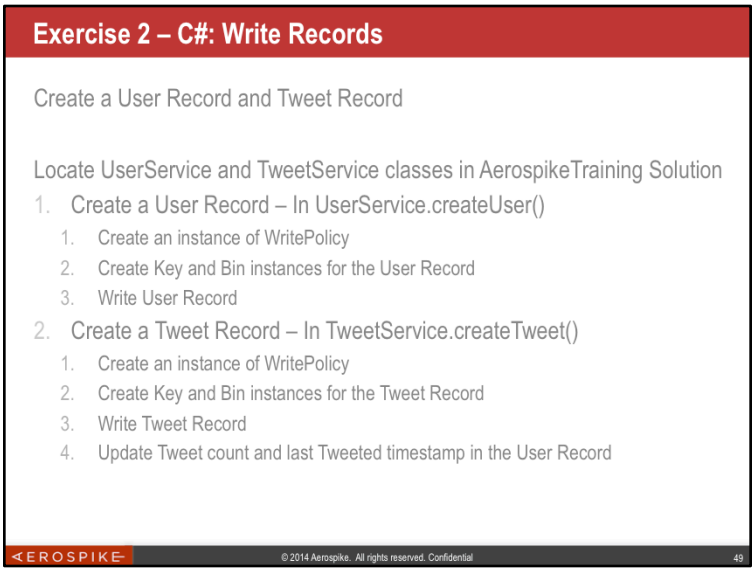
1. In the Main() method for the class Program, add code similar to this;

```
// Specify IP of one of the nodes in the cluster
string asServerIP = "172.16.159.206";
// Specity Port that the node is listening on
int asServerPort = 3000;
// Establish connection
client = new AerospikeClient(asServerIP, asServerPort);
```

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

2. At the end of Main() method, 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.Connected)
{
     // Close Aerospike server connection
     client.Close();
}
```



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 = Bin.AsList("interests",
           interests.Split(',').ToList<object>());
```

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. Create a WritePolicy

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

2. Create Key and Bin instances

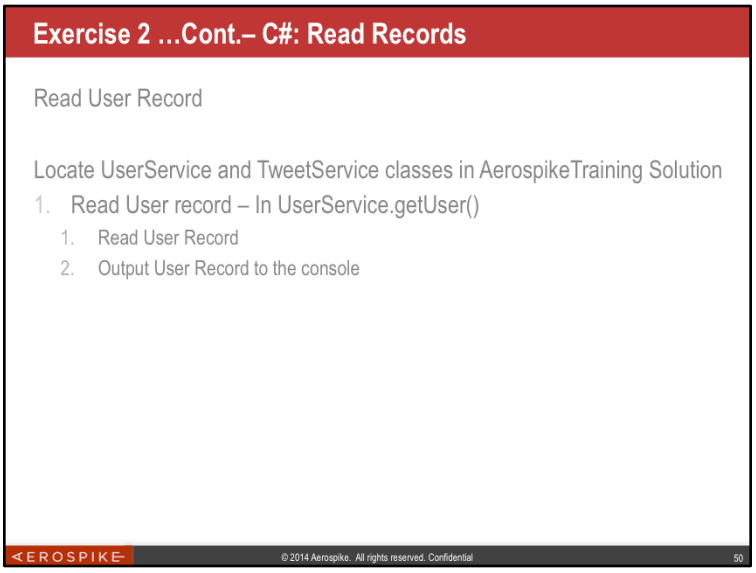
```
long ts = getTimeStamp();
tweetKey = new Key("test", "tweets", username + ":"
                                       + nextTweetCount);
Bin bin1 = new Bin("tweet", tweet);
Bin bin2 = new Bin("ts", ts);
Bin bin3 = new Bin("username", username);
```

3. Write a tweet record using the Key and Bins

```
client.Put(wPolicy, tweetKey, bin1, bin2, bin3);
Console.WriteLine("\nINFO: Tweet record created!");
```

4. Update the user record with tweet count

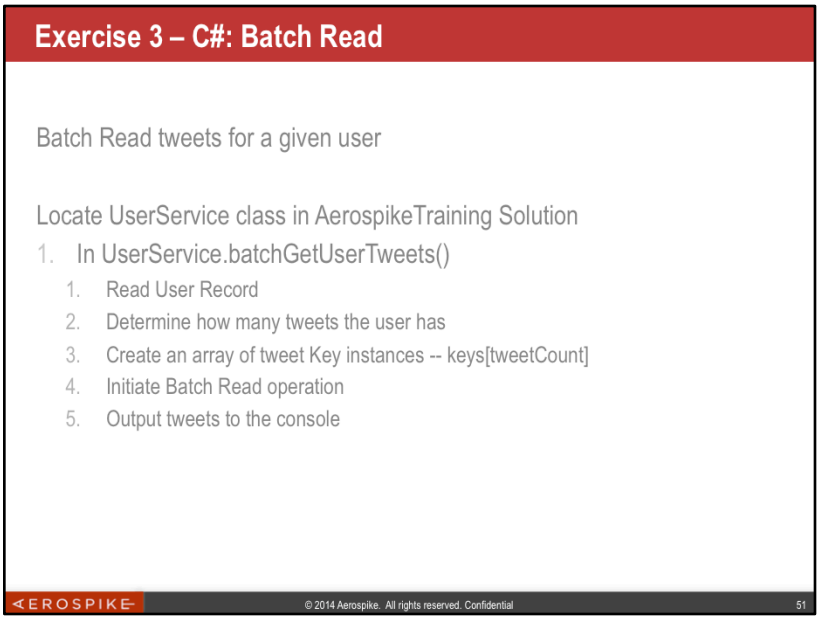
```
// Update tweet count and last tweet'd timestamp
// in the user record
updateUser(client, userKey, wPolicy, ts, nextTweetCount);
```



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.WriteLine("\nINFO: User record read successfully! Here are the details:\n");
    Console.WriteLine("username:     " + userRecord.GetValue("username"));
    Console.WriteLine("password:     " + userRecord.GetValue("password"));
    Console.WriteLine("gender:       " + userRecord.GetValue("gender"));
    Console.WriteLine("region:       " + userRecord.GetValue("region"));
    Console.WriteLine("tweetcount:   " + userRecord.GetValue("tweetcount"));
    List<object> interests = (List<object>) userRecord.GetValue("interests");
    Console.WriteLine("interests:    " + interests.Aggregate((x, y) => x + "," + y));
}
else
{
    Console.WriteLine("ERROR: User record not found!");
}
```



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 = int.Parse(userRecord.GetValue("tweetcount").ToString());
```

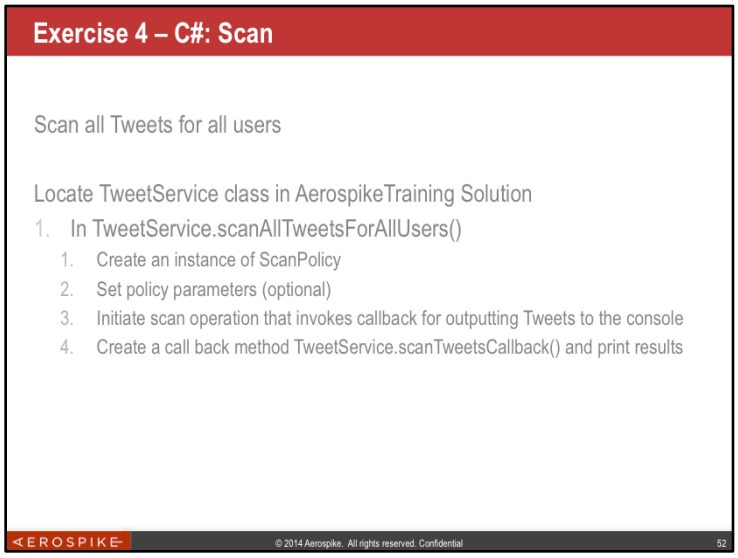
3. Create a "list" of tweet keys

```
// Create an array of keys so we can initiate batch read operation
Key[] keys = new Key[tweetCount];
for (int i = 0; i < keys.Length; i++)
{
    keys[i] = new Key("test", "tweets", (username + ":" + (i + 1)));
}
Console.WriteLine("\nHere's " + username + "'s tweet(s):\n");
```

4. Perform a Batch operation to read all the tweets

```
// Initiate batch read operation
Record[] records = client.Get(null, keys);
for (int j = 0; j < records.Length; j++)
{
    Console.WriteLine(records[j].GetValue("tweet"));
}
```

5. Then print out the tweets



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

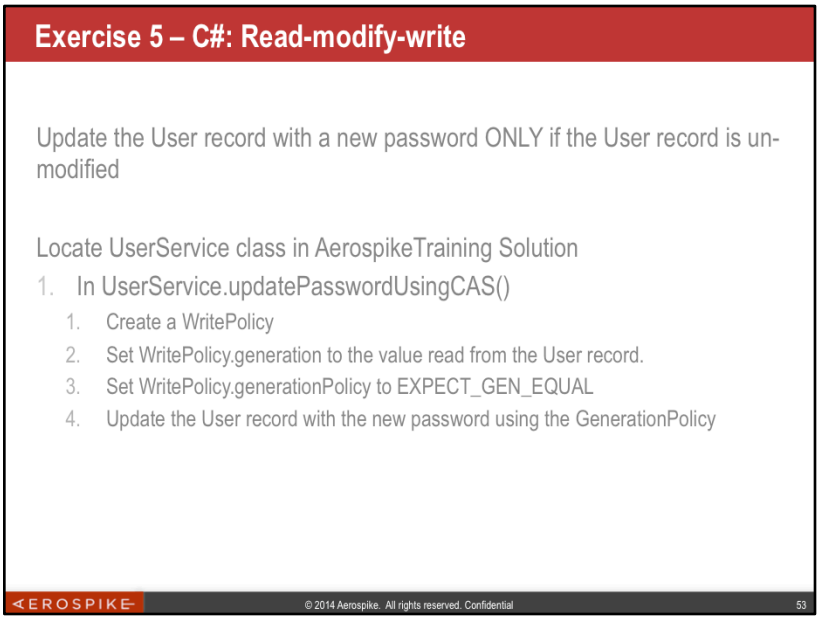
```
ScanPolicy policy = new ScanPolicy();
policy.concurrentNodes = true;
policy.priority = Priority.LOW;
policy.includeBinData = true;
```

3. Perform a Scan operation

```
client.ScanAll(policy, "test", "tweets", scanTweetsCallback, "tweet");
```

4. Create a call back method TweetService.scanTweetsCallback() and print the results

```
public void scanTweetsCallback(Key key, Record record)
{
     Console.WriteLine(record.GetValue("tweet"));
} //scanTweetsCallback
```



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

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.WriteLine("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", password);
    client.Put(writePolicy, userKey, passwordBin);

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

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()

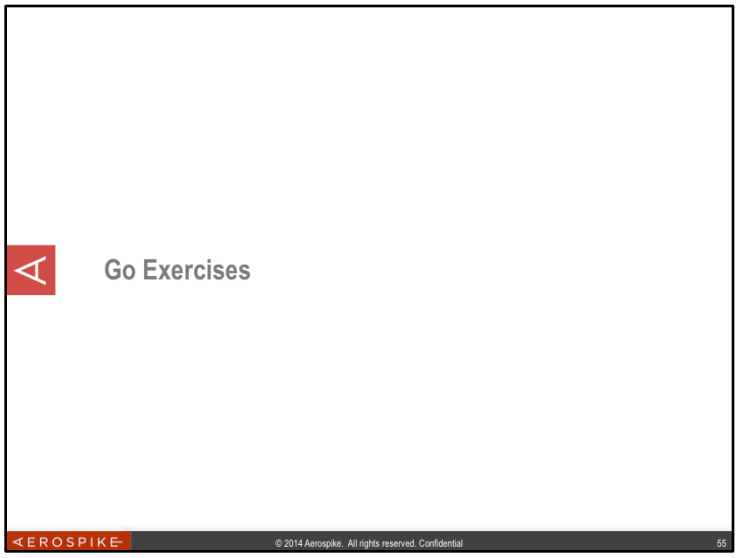
1. Comment out the code added in exercise 2

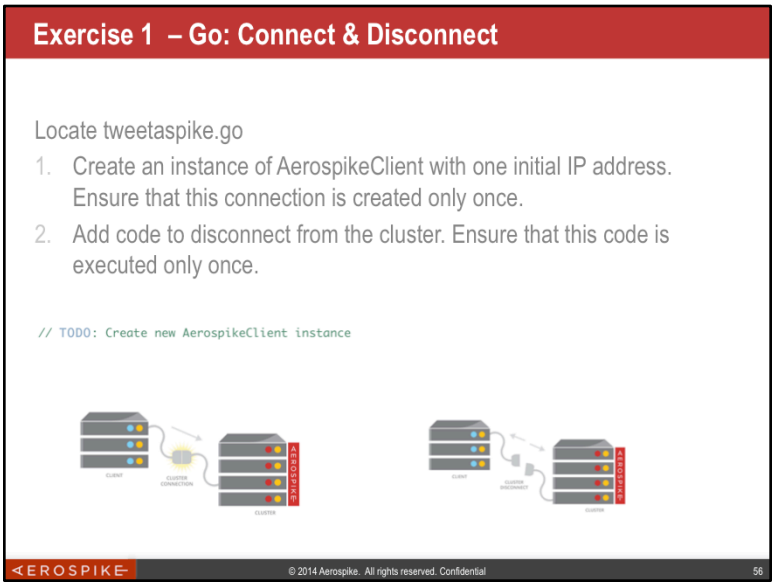
2. Uncomment the line:

```
// TODO: Update tweet count and last tweeted timestamp in the user record using Operate
// Exercise 6
// updateUserUsingOperate(client, userKey, policy, ts);
```

3. 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());
Console.WriteLine("INFO: The tweet count now is: " + record.GetValue("tweetcount"));
```


Go Exercises
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
55



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 main() function add code similar to this;

```
fmt.Println("INFO: Connecting to Aerospike cluster...")
// Establish connection to Aerospike server
client, err := NewClient("54.90.203.181", 3000)
panicOnError(err)
```

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

2. Add a “defer” and call Close() to disconnect from the cluster. This should only be done once. After close() is called, the client instance cannot be used.

```
defer client.Close()
```

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

1. Create a WritePolicy

```
// Write record
wPolicy := NewWritePolicy(0, 0) // generation = 0, expiration = 0
wPolicy.RecordExistsAction = UPDATE
```

2. Create Key and Bin instances

```
key, _ := NewKey("test", "users", username)
bin1 := NewBin("username", username)
bin2 := NewBin("password", password)
bin3 := NewBin("gender", gender)
bin4 := NewBin("region", region)
bin5 := NewBin("lasttweeted", 0)
bin6 := NewBin("tweetcount", 0)
arr := strings.Split(interests, ",")
bin7 := NewBin("interests", arr)
```

1. Write a user record using the Key and Bins

```
err := client.PutBins(wPolicy, key, bin1, bin2, bin3,
                            bin4, bin5, bin6, bin7)
panicOnError(err)
```

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

1. Create a WritePolicy

```
// Write record
wPolicy := NewWritePolicy(0, 0) // generation = 0, expiration = 0
wPolicy.RecordExistsAction = UPDATE
// Create timestamp to store along with the tweet so we can
// query, index and report on it
timestamp := getTimeStamp()
```

2. Create Key and Bin instances

```
keyString := fmt.Sprintf("%s:%d", username, tweetCount)
tweetKey, _ := NewKey("test", "tweets", keyString)
bin1 := NewBin("tweet", tweet)
bin2 := NewBin("ts", timestamp)
bin3 := NewBin("username", username)
```

3. Write a tweet record using the Key and Bins

```
err := client.PutBins(wPolicy, tweetKey, bin1, bin2, bin3)
panicOnError(err)
fmt.Printf("\nINFO: Tweet record created! with key: %s, %v, %v, %v\n",
                        keyString, bin1, bin2, bin3)
```

4. Update the user record with tweet count

```
// Update tweet count and last tweet'd timestamp in the user record
updateUser(client, userKey, nil, timestamp, tweetCount)
```

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

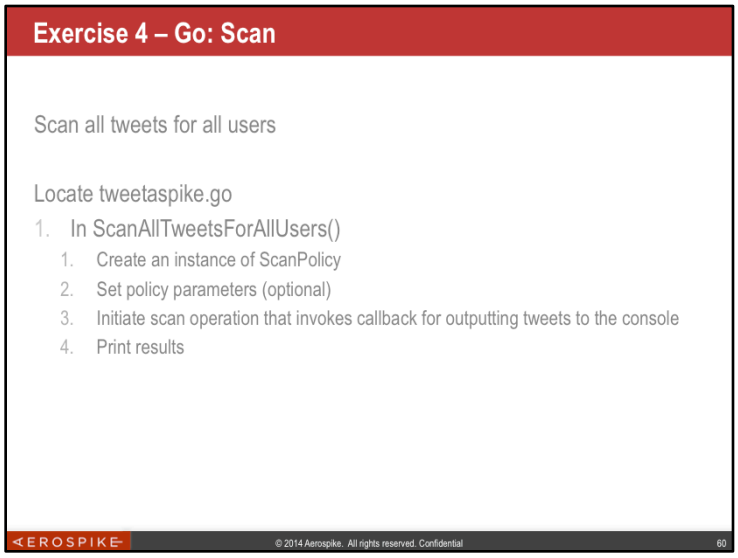
1. Read a User record
2. Output the User record to the console

```
// Check if username exists
userKey, _ := NewKey("test", "users", username)
userRecord, err := client.Get(nil, userKey)
panicOnError(err)
if userRecord != nil {
      fmt.Printf("\nINFO: User record read successfully! Here are the details:\n")
      fmt.Printf("username:   %s\n", userRecord.Bins["username"].(string))
      fmt.Printf("password:   %s\n", userRecord.Bins["password"].(string))
      fmt.Printf("gender:     %s\n", userRecord.Bins["gender"].(string))
      fmt.Printf("region:     %s\n", userRecord.Bins["region"].(string))
      fmt.Printf("tweetcount: %d\n", userRecord.Bins["tweetcount"].(int))
      fmt.Printf("interests:  %v\n", userRecord.Bins["interests"])
} else {
      fmt.Printf("ERROR: User record not found!\n")
}
```

Read all the tweets for a given user. In BatchGetUserTweets(), add code similar to this:

1. Read a user record
2. Get the tweet count
3. Create a "list" of tweet keys
1. Perform a Batch operation to read all the tweets
2. Then print out the tweets

```
userKey, _ := NewKey("test", "users", username)
userRecord, err := client.Get(nil, userKey)
panicOnError(err)

if userRecord != nil {
      // Get how many tweets the user has
      tweetCount := userRecord.Bins["tweetcount"].(int)

      // Create an array of keys so we can initiate batch read
      // operation
      keys := make([]*Key, tweetCount)

      for i := 0; i < len(keys); i++ {
            keyString, _ := fmt.Scanf("%s:%d", username, i+1)
            key, _ := NewKey("test", "tweets", keyString)
            keys[i] = key
      }

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

      // Initiate batch read operation
      if len(keys) > 0 {
            records, err := client.BatchGet(NewPolicy(), keys)
            panicOnError(err)
            for _, element := range records {
                  fmt.Println(element.Bins["tweet"])
            }
      }
}
```



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

In the ScanAllTweetsForAllUsers(), add code similar to this:

1. Create a ScanPolicy
2. Set policy parameters

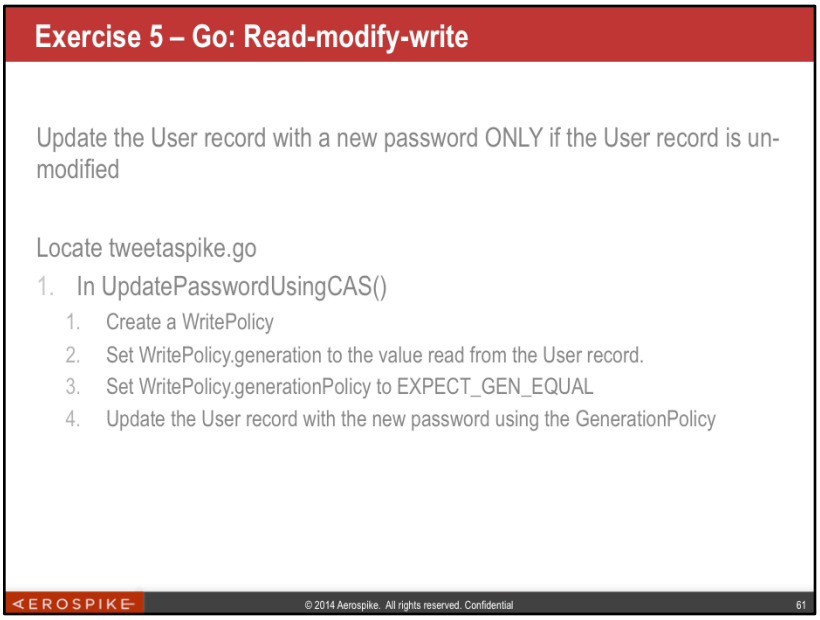
```
policy := NewScanPolicy()
policy.ConcurrentNodes = true
policy.Priority = LOW
policy.IncludeBinData = true
```

3. Perform a Scan operation

```
records, err := client.ScanAll(policy, "test", "tweets", "tweet")
panicOnError(err)
```

4. Print the results

```
for element := range records.Records {
     fmt.Println(element.Bins["tweet"])
}
```

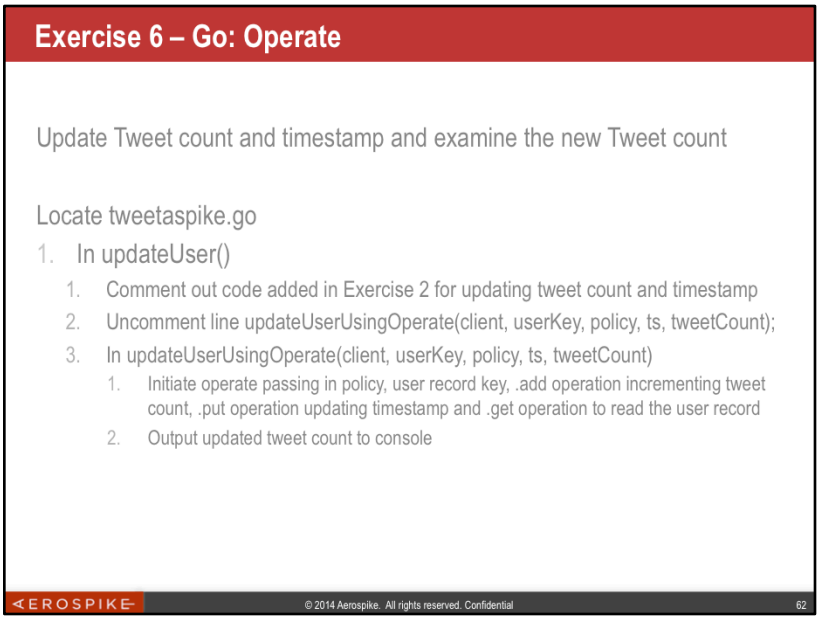


Update the User record with a new password ONLY if the User record is un-modified

In 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, _ := NewKey("test", "users", username)
userRecord, err := client.Get(nil, userKey)
panicOnError(err)
if err == nil {
      // Get new password
      var password string
      fmt.Print("Enter new password for %s:", username)
      fmt.Scanf("%s", &password)

      writePolicy := NewWritePolicy(0, 0) // generation = 0, expiration = 0
      // record generation
      writePolicy.Generation = userRecord.Generation
      writePolicy.GenerationPolicy = EXPECT_GEN_EQUAL
      // password Bin
      passwordBin := NewBin("password", password)
      err = client.PutBins(writePolicy, userKey, passwordBin)
      panicOnError(err)
      fmt.Printf("\nINFO: The password has been set to: %s", password)
} else {
      fmt.Printf("ERROR: User record not found!")
}
```



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 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);
```

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

```
record, err := client.Operate(policy, userKey,
     AddOp(NewBin("tweetcount", 1)),
     PutOp(NewBin("lasttweeted", timestamp)),
     GetOp())
panicOnError(err)

fmt.Printf("\nINFO: The tweet count now is: %d\n",
                    record.Bins["tweetcount"])
```

Summary

You have learned how to:

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


AEROSPIKE
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
64