Aerospike for Developers

Key-value Operations

AEROSPIKE

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

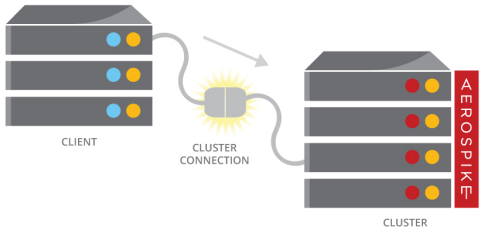
Connecting to a Cluster

Connecting to a Cluster

Connecting to a Cluster is similar to connecting to other database technologies, the **difference** is you are connecting to a Cluster, not a server

At the start of your process (program) create a client instance once, Supplying one or more node addresses.

The node addresses are used by the Client to discover ALL the nodes in the Cluster.

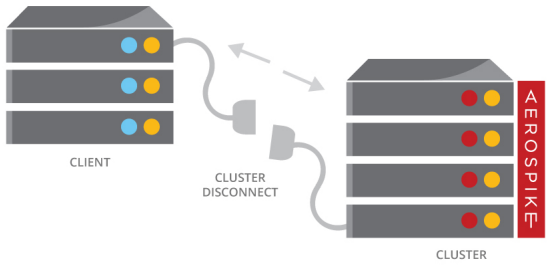



```
// C# connection to a cluster
string asServerIP = "172.16.159.152";
int asServerPort = 3000;
AerospikeClient client = new AerospikeClient(asServerIP, asServerPort);

// Go connection to a cluster
client, err := NewClient("54.163.76.64", 3000)

// Java connection using multiple nodes
Host[] hosts = new Host[] {
            new Host("a.host", 3000),
            new Host("another.host", 3000),
            new Host("and.another.host", 3000)
};
AerospikeClient client = new AerospikeClient(new ClientPolicy(), hosts);
```

Disconnecting

At the **end of the process**, disconnect the Client instance from the cluster by calling close method. The close method will:

- Close sockets
- Free object references
- Terminates the client monitor thread
- Remove the client thread pool




```
// C# Close
client.Close();
```

```
// Java Close
client.close();
```

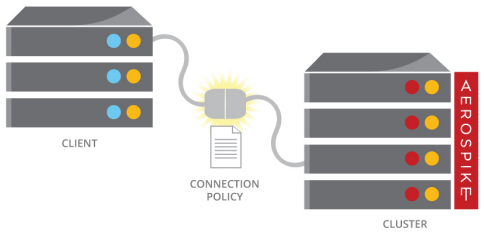
```
// Go Close
defer client.Close()
```

Client Policy

An optional ClientPolicy modifies Aerospike's default behavior when connecting to a cluster.

ClientPolicy Attributes:

- Connection timeout
- Fail if not connected
- Max Socket Idle
- Max Threads




```
// C# connection with a Client Policy
ClientPolicy cPolicy = new ClientPolicy();
cPolicy.maxThreads = 200; //200 threads
cPolicy.maxSocketIdle = 3; // 3 seconds
AerospikeClient client = new AerospikeClient(cPolicy,
            "172.16.159.152", 3000);
```

```
// Java connection with Client policy
ClientPolicy clientPolicy = new ClientPolicy();
clientPolicy.maxThreads = 200; //200 threads
clientPolicy.maxSocketIdle = 3; // 3 seconds
AerospikeClient client = new AerospikeClient(clientPolicy,
            "a.host", 3000);
```

Recommendation: Use the defaults

Write Operations

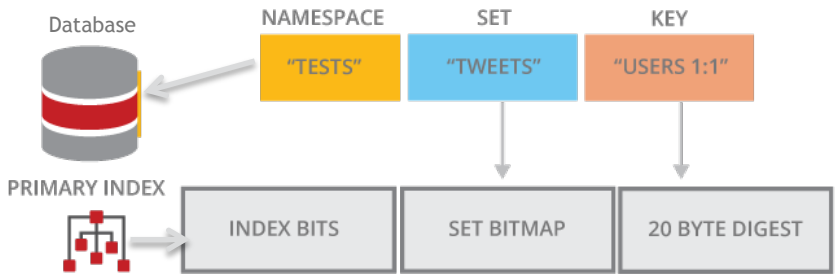
Key

Each key-value operation uses one or more keys. A **Key** is a **structure** containing:

- Namespace (database) - String representing the Namespace name
- Set (table) – String representing the Set name, if omitted records will be in the "null" Set
- Key – The value of the primary key

  OR

- Digest – 20 byte hash of the key value



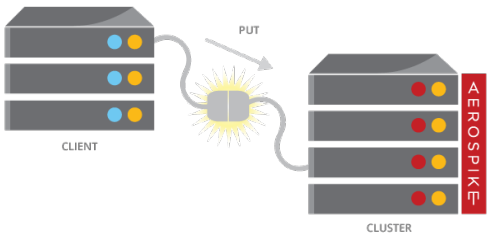

```
// C#
new Key("test", "users", username);

// Java
new Key("test", "users", username);
```

```
// Go
key, _ := NewKey("test", "users", username)
```

Write a Record

With the **put** operation, you can write:

- Multiple bins (columns)
- A Single Bin
- Delete a Bin
- A Time-to-Live (TTL)
- Timeout



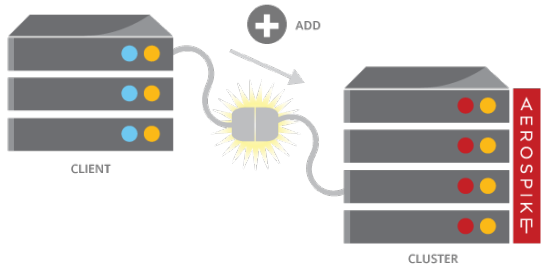

```java
// Java write
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.expiration = 5; // Time To live of 5 seconds

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

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

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

key, _ := NewKey("test", "users", username)
bin1 := NewBin("username", username)
bin2 := NewBin("password", password)

err := client.PutBins(wPolicy, key, bin1, bin2)
panicOnError(err)
```

Add

The **add** operation increments an integer Bin by a specific value. You can supply a negative value to decrement. An integer Bin is 8 bytes unsigned.

Use case: Counters



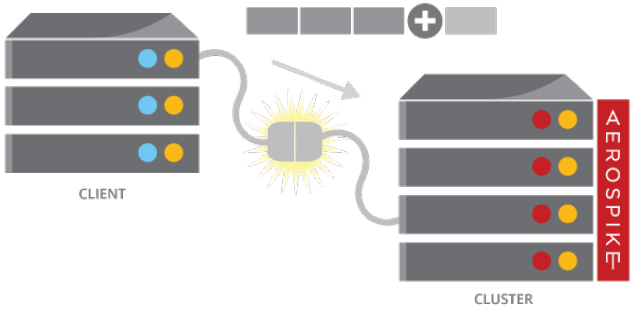

```
// C# Add
Key userKey = new Key("test", "users", "user1234");
Bin counter = new Bin("tweetcount", 1);
client.Add(null, userKey, counter);

// Java add
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
Bin counter = new Bin("tweetcount", 1);
client.add(wPolicy, key, counter);
```

Append - Prepend

The **append/prepend** operations work on String or byte arrays (BLOB) bins. The **append** operation adds a string/byte array to the end of the bin value. The **prepend** operation adds a string/byte array to the start of the bin value.

If the bin does not exist,

it will be created.

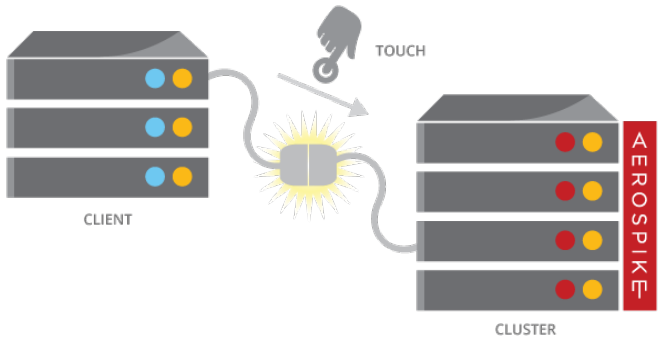



```
// C# Append
Key userKey = new Key("test", "users", "user1234");
Bin bin2 = new Bin("greet", " world");
client.Append(null, userKey, bin2);

// Java append
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", "user1234",);
Bin bin2 = new Bin("greet", " world");
client.append(wPolicy, key, bin2);

// C# Prepend
Key userKey = new Key("test", "users", "user1234");
Bin bin2 = new Bin("greet", "hello ");
client.Prepend(null, userKey, bin2);

// Java prepend
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", "user1234",);
Bin bin2 = new Bin("greet", "hello ");
client.prepend(wPolicy, key, bin2);
```

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.

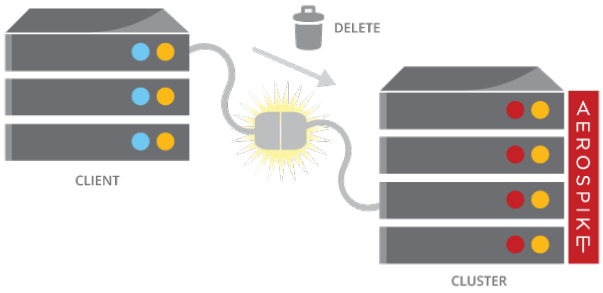
A multi–record operation is **not** atomic.




```
// 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);
```

Delete

The **delete** operation deletes the entire record from the database.




```
// C# Delete record.
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.Delete(wPolicy, key);
```

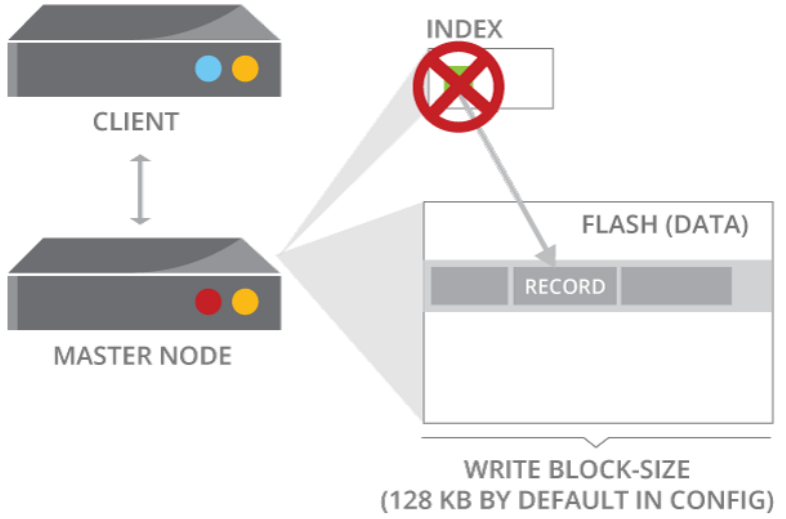
```
// Java Delete record.
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.delete(wPolicy, key);
```

Delete …

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

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

Note: Aerospike does not "tombstone" deleted records.



WritePolicy

A **WritePolicy** modifies the behavior of a write operation. If omitted, default values will be used.

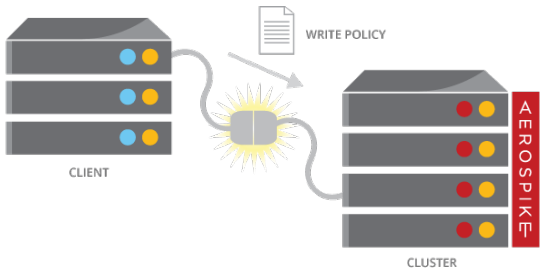
WritePolicy attributes:

- Timeout
- Expiration
- Generation
- Generation policy
- Record exists action

```
// C# write policy
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.timeout = 50; // 50 milliseconds
```

```
// Java write policy
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.timeout = 50; // 50 milliseconds
```

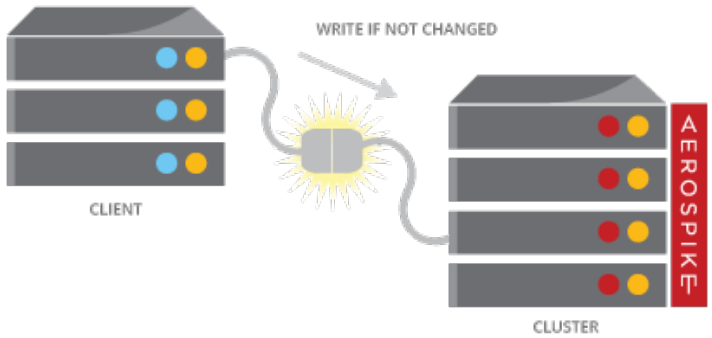
The WritePolicy is quite powerful. We will cover how to use these attributes in detail.



Generation Policy – WritePolicy

A **record generation** indicates how many times the record has been written and is **incremented** on each write.

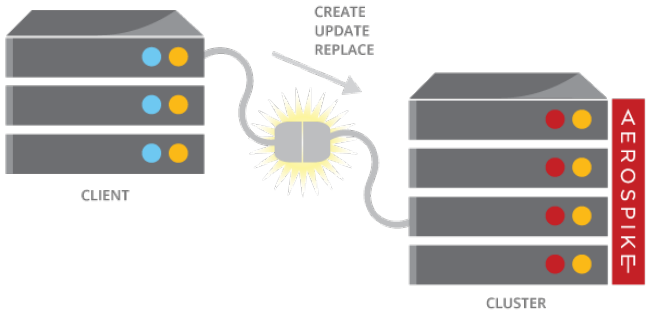
- NONE
  - Do not use a record generation to restrict writes. (Default)
- *EXPECT_GEN_EQUAL*
  - Update/delete record if expected generation is equal to server generation. Otherwise, fail.



Record Exists – WritePolicy

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

- UPDATE
  - Create or update a record
- UPDATE_ONLY
  - Update record only. Fail if record does not exist.
- REPLACE
  - Create or replace record. Delete previous Bins.
- REPLACE_ONLY
  - Replace record only. Fail if record does not exist.
- CREATE_ONLY
  - Create only. Fail if record exists.

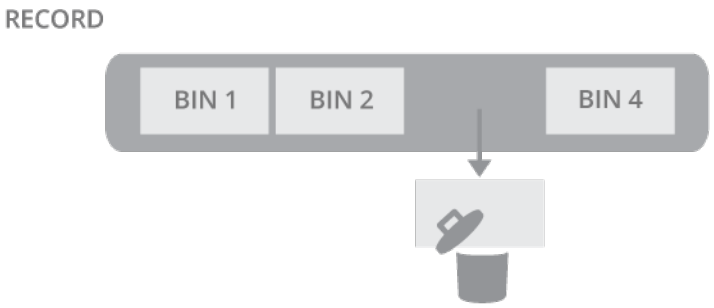


Deleting a Bin

In a **put** operation, you can delete one or more Bins by setting the Bin value as null:

```
// C# delete a bin
WritePolicy wPolicy = new WritePolicy();
Bin bin1 = Bin.asNull("shoe-size"); // Set bin value to null to drop bin.
client.Put(wPolicy, key, bin1);
```

```
// Java delete a bin
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
Bin bin1 = Bin.asNull("shoe-size"); // Set bin value to null to drop bin.
client.put(wPolicy, key, bin1);
```



Read Operation

Read a Record

The **get** operation returns a Record (row).

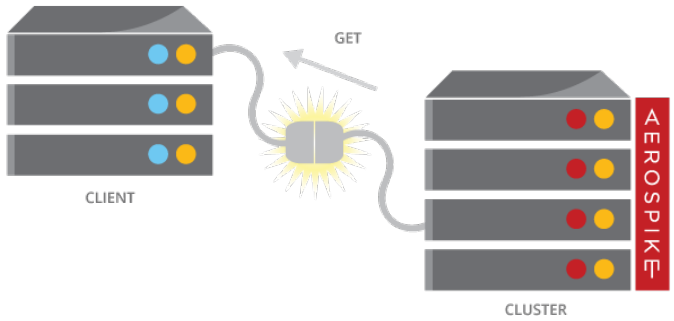
You can read:

- All bins (columns)
- Specific bins

A **Record** consists of:

- Generation
- Expiration
- Bins

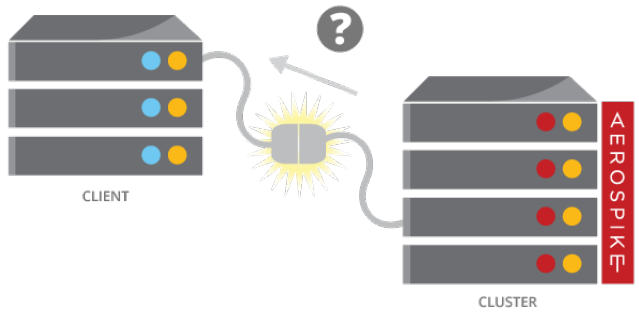
```
// C#
Key userKey = new Key("test", "users", username);
Record record = client.Get(null, userKey);

// Java read specific bins
Key key = new Key("test", "users", userName);
Record record = this.client.get(null, key,
          "username",
          "password",
          "gender",
          "region");

// Java get header data
Key key = new Key("test", "users", userName);
Record record = this.client.getHeader(null, key);
```



Exists

The **exists** operation is a light weight alternative to reading a Record to check if it exists.

Note: The Record could "not exist" shortly after the operation.

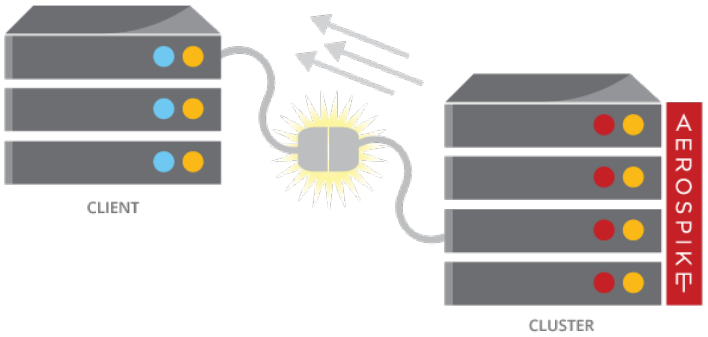



```
// C# Exists
Key userKey = new Key("test", "users", "user1234");
bool recordKeyExists = client.Exists(null, userKey);

// Java exists
Key key = new Key("test", "users", username);
boolean itsHere =  client.exists(policy, key);
```

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);
```

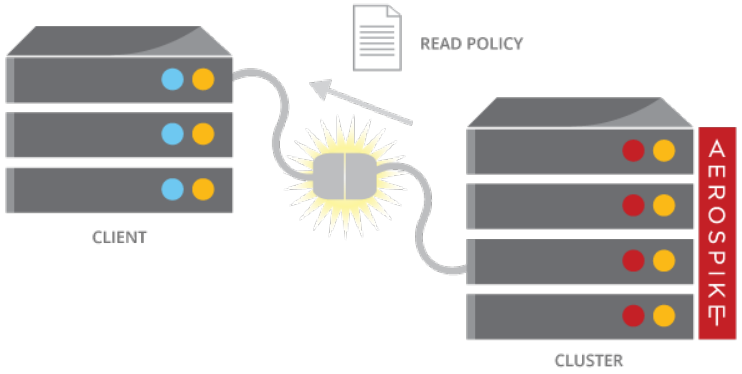
Read Policy

When executing a Read operation, you can ***optionally*** supply a **Policy** object to the method. If omitted, default values will be used.

- Policy
  - Timeout
  - Maximum retries
  - Sleep between retries
  - Priority

```
// C# write policy
Policy policy = new Policy();
policy.timeout = 50; // 50 milliseconds
```

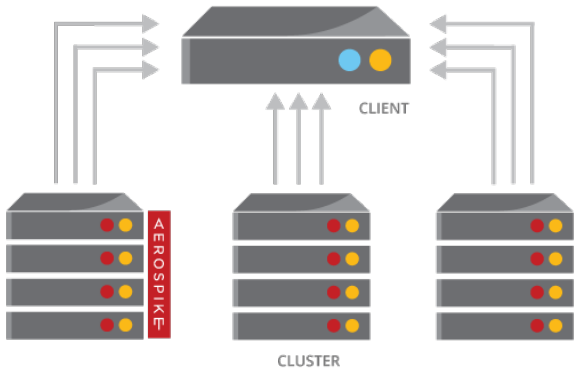
```
// Java policy
Policy policy = new Policy();
policy.timeout = 50; // 50 milliseconds
```



Scan

The **Scan** operation, similar to a table scan, returns all Records in a specified namespace and, optionally, a Set.

```
// Java Scan
client.scanAll(null, "test", "tweets", new ScanCallback() {

      @Override
      public void scanCallback(Key key, Record record)
                              throws AerospikeException {
                  System.out.println(record.getValue("tweet"));
      }
}, "tweet");


// C# Scan
client.ScanAll(null, "test", "tweets",
                        scanTweetsCallback, "tweet");

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



Scan Policy

A **ScanPolicy** can optionally be supplied to modify the Scan operation.

Along with all the attributes of Policy, it contains additional attributes that define the properties of the Scan. These are:

- Concurrent nodes – Default: true
- Fail on cluster change – Default: false
- Include bin data – Default: true
- Maximum concurrent nodes – Default: All nodes in the cluster

```
// C# Scan policy
ScanPolicy policy = new ScanPolicy();
policy.concurrentNodes = true;
policy.includeBinData = true;
client.ScanAll(policy, . . .);
```

```
// Java Scan policy
ScanPolicy policy = new ScanPolicy();
policy.concurrentNodes = true;
policy.includeBinData = true;
client.scanAll(policy, . . .);
```

Key-value Techniques

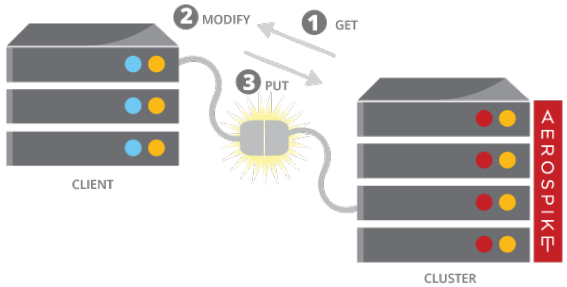
Read Modify Write (CAS)

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);
```

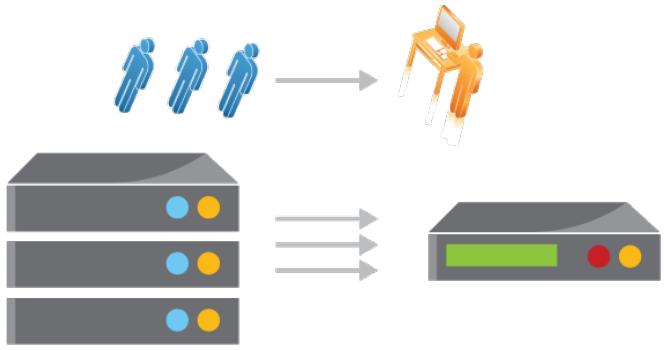
Hot Keys

Hot keys are records that are **frequently** accessed

- How hot is hot?
- Queue for a Record
- Result code 14 – KEY_BUSY
- Configuration sets queue size
  - transaction-pending-limit
  - Default: 20

How to **mitigate** Hot Keys

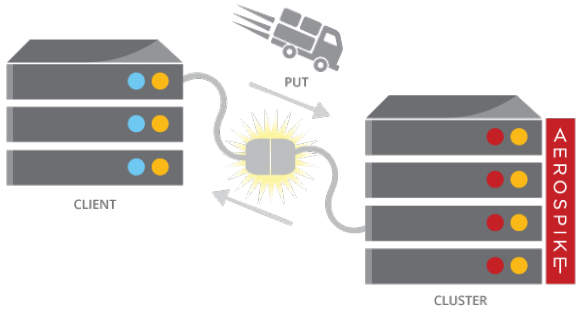
- Use RAM namespace
- Use multiple Records
  - Batch read
  - Aggregate in client



Operate

Multiple operations on the same Record

- Update and Select in one operation
- Read and Write operations
- Read operations are done last
- Example: add to a counter and read the value




```
// C# Operate
client.Operate(policy, userKey,
      Operation.Put(new Bin("tweetcount", 153)),
      Operation.Put(new Bin("lasttweeted", 1406755079L)));

// Java operate
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.operate(wPolicy, key,
      Operation.put(new Bin("tweetcount", 153)),
      Operation.put(new Bin("lasttweeted", 1406755079L)));
```

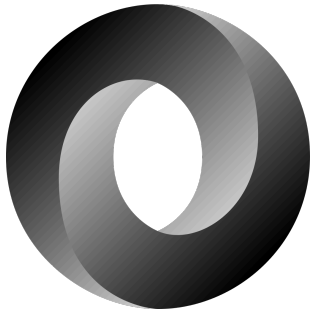
List, Maps and JSON

Lists and **Maps** (Dictionaries) are supported in most modern languages. The Client API translates the language types to the database type of List or Map respectively.

JSON is either a List containing Lists and Maps, or a Map containing Lists and Maps. JSON is easily stored in a Bin Type of:

- List
- Map

```
// Java JSON example
Bin bin7 = Bin.asMap("interests", new JSONObject(...));
```



Single Bin

A namespace can be configured, and optimized, for a **single Bin record**.

- No Bin management overhead
- Integer – stored in Index for free (data-in-index)

Operations

- Add
- Append

Cannot be used with a secondary index

Errors and Result Codes

Error Handling

When an error condition occurs the client will throw an **AerospikeException** which contains:

- A ResultCode
- A Message

```
// Java error handling
try {

      . . .

} catch (AerospikeException e) {
      System.out.println("EXCEPTION - Message: "
            + e.getMessage());
      System.out.println("EXCEPTION – Result code: "
            + e.getResultCode());
}
```

```
// C# error handling
try
{

      . . .

}
catch (AerospikeException e)
{
      Console.WriteLine("EXCEPTION - Message: "
            + e.Message);
}
```

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

Lab: Key-value Operations

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

Lab Overview

The lab exercises add functionality to a simple Twitter- like console application (tweetaspike) using Aerospike as the database.

In this Lab, we will focus on key-value operations and techniques. You will add code to:

- Create users and Tweets
- Read all the Tweets for a user
- Use an advance Key-value feature

The exercises shell is located on your USB stick or in your “unzipped” directory

```
~/exercise/Key-valueOperations/<language>
```

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

Tweetaspike Data Model

Users

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

Tweets

- Namespace: test, Set: tweets, Key: <username:<counter>>
- Bins:
  - tweet – string
  - ts - int (Stores epoch timestamp of the tweet)
  - username - string

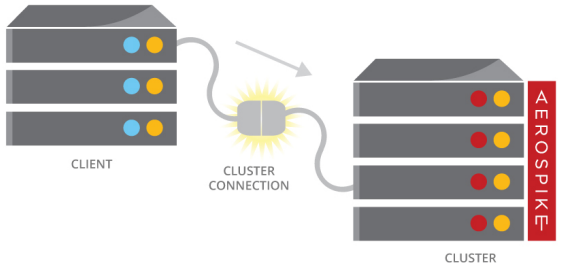
Java Exercises

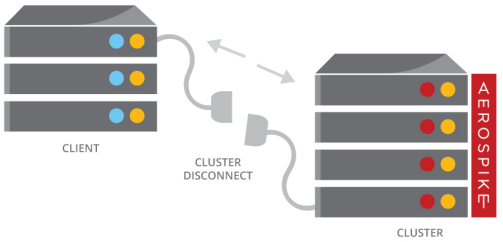
Exercise 1 – Java: Connect & Disconnect

Locate Program Class in AerospikeTraining Solution

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

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





Exercise 2 – Java: Write Records

Create a User Record and Tweet Record

Locate UserService and TweetService classes in AerospikeTraining Solution

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

Exercise 2 …Cont.– Java: Read Records

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

Locate UserService and TweetService classes in AerospikeTraining Solution

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

Exercise 3 – Java: Batch Read

Batch Read Tweets for a given user

Locate UserService class in AerospikeTraining Solution

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

Exercise 4 – Java: Scan

Scan all tweets for all users

Locate TweetService class in AerospikeTraining Solution

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

Exercise 5 – Java: Read-modify-write

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

Locate UserService class in AerospikeTraining Solution

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

Exercise 6 – Java: Operate

Update Tweet count and timestamp and examine the new Tweet count

Locate TweetService class in AerospikeTraining Solution

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

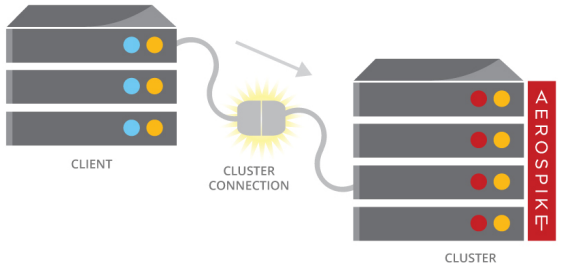
C# Exercises

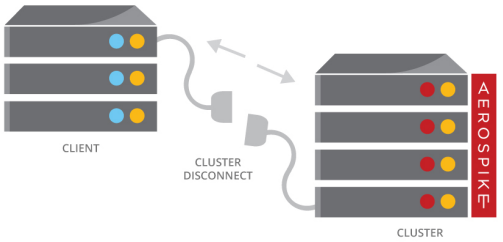
Exercise 1 – C#: Connect & Disconnect

Locate Program Class in AerospikeTraining Solution

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

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





Exercise 2 – C#: Write Records

Create a User Record and Tweet Record

Locate UserService and TweetService classes in AerospikeTraining Solution

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

Exercise 2 …Cont.– C#: Read Records

Read User Record

Locate UserService and TweetService classes in AerospikeTraining Solution

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

Exercise 3 – C#: Batch Read

Batch Read tweets for a given user

Locate UserService class in AerospikeTraining Solution

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

Exercise 4 – C#: Scan

Scan all Tweets for all users

Locate TweetService class in AerospikeTraining Solution

1. In TweetService.scanAllTweetsForAllUsers()
    1. Create an instance of ScanPolicy
    2. Set policy parameters (optional)
    3. Initiate scan operation that invokes callback for outputting Tweets to the console
    4. Create a call back method TweetService.scanTweetsCallback() and print results

Exercise 5 – C#: Read-modify-write

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

Locate UserService class in AerospikeTraining Solution

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

Exercise 6 – C#: Operate

Update Tweet count and timestamp and examine the new Tweet count

Locate TweetService class in AerospikeTraining Solution

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

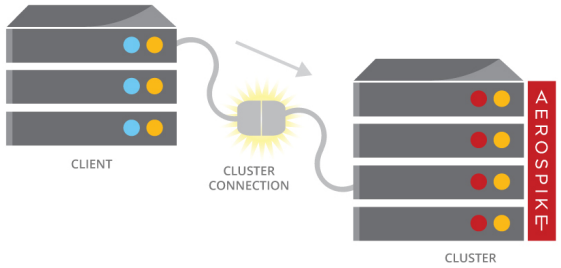
Go Exercises

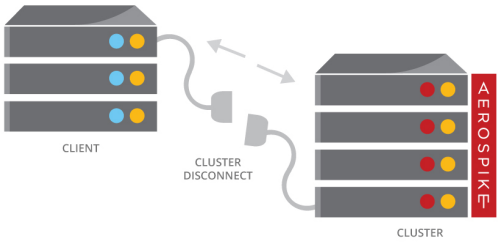
Exercise 1 – Go: Connect & Disconnect

Locate tweetaspike.go

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

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





Exercise 2 – Go: Write Records

Create a User Record and Tweet Record

Locate tweetaspike.go

1. Create a User Record – In CreateUser()
    1. Create an instance of WritePolicy
    2. Create Key and Bin instances for the User Record
    3. Write User Record
2. Create a Tweet Record – In CreateTweet()
    1. Create an instance of WritePolicy
    2. Create Key and Bin instances for the Tweet Record
    3. Write Tweet Record
    4. Update tweet count and last tweeted timestamp in the User Record

Exercise 2 …Cont.– Go: Read Records

Read User Record

Locate tweetaspike.go

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

Exercise 3 – Go: Batch Read

Batch Read tweets for a given user

Locate tweetaspike.go

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

Exercise 4 – Go: Scan

Scan all tweets for all users

Locate tweetaspike.go

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

Exercise 5 – Go: Read-modify-write

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

Locate tweetaspike.go

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

Exercise 6 – Go: Operate

Update Tweet count and timestamp and examine the new Tweet count

Locate tweetaspike.go

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

Summary

You have learned how to:

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