1. Log on to Riak server I have created on the cloud

Open Putty (need latest version download 0.7) and add the following details:

Hostname: [ubuntu@xx.xx.xx.xx](mailto:ubuntu@xx.xx.xx.xx) (today: 54.226.53.50**)**

Click SSH/Auth and in the private key file download the pkk file from the lab folder for today (NoSQL16.pkk) and save it on C:/ and browse to it.

Click Open

When you are logged on I want you to change to be a different user using (your t number)

sudo useradd Tnumber

Now become the user with:

sudo su Tnumber

Starting a Cluster (I will do this – not you!)

Verify your servers are healthy by checking their stats using

curl <http://localhost:10018/stats>

It should look something like this (edited for readability):

{  
 "vnode\_gets":0,  
 "vnode\_puts":0,  
 "vnode\_index\_reads":0,  
 ...  
 "connected\_nodes":[

*"dev2@127.0.0.1"*,

*"dev3@127.0.0.1"*

],  
 ...  
 "ring\_members":[

*"dev1@127.0.0.1"*, *"dev2@127.0.0.1"*, *"dev3@127.0.0.1"*

],

"ring\_num\_partitions":64,  
"ring\_ownership":

*"[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]"*, ...

1. Run a REST query

Let’s issue a bad query. -I tells cURL that we want only the header response.

**curl -I** [**http://localhost:10018/riak/no\_bucket/no\_key**](http://localhost:8091/riak/no_bucket/no_key)

1. Put in some data (use your initials for the bucket names throughout these exercises)

e.g. /riak/pgfavs/db. The easiest way is copy and paste the first line only and edit it, then copy and paste the other lines.

**curl -v -X PUT** [**http://localhost:10018/riak/pggfavs/db**](http://localhost:10018/riak/pggfavs/db) **\  
 -H "Content-Type: text/html" \  
 -d "<html><body><h1>My new favorite DB is RIAK</h1></body></html>"**

If you curl <http://localhost:10018/riak/pggfavs/db> , you’ll get a nice message from yourself.

1. A bucket of animals

**curl -v -X PUT** [**http://localhost:10018/riak/pgganimals/ace**](http://localhost:10018/riak/pgganimals/ace) **\  
 -H "Content-Type: application/json" \  
 -d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}'**

We can view our list of buckets that have been created.

**curl -X GET** [**http://localhost:10018/riak?buckets=true**](http://localhost:8091/riak?buckets=true)

Optionally, you can return the set results with the ?returnbody=true parameter, which we’ll test by adding another animal, Polly:

**curl -v -X PUT http://localhost:10018/riak/pgganimals/polly?returnbody=true \  
 -H "Content-Type: application/json" \  
 -d '{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}'**

If we aren’t picky about our key name, Riak will generate one when using POST.

**curl -i -X POST** [**http://localhost:10018/riak/pgganimals**](http://localhost:10018/riak/pgganimals)**/ \  
 -H "Content-Type: application/json" \  
 -d '{"nickname" : "Sergeant Stubby", "breed" : "Terrier"}'**

The generated key will be in the header under Location—also note the 201 success code in the header.

HTTP/1.1 201 Created  
Vary: Accept-Encoding  
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)  
Location: /riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO  
Date: Tue, 05 Apr 2011 07:45:33 GMT  
Content-Type: application/json  
Content-Length: 0

A GET request (cURL’s default if left unspecified) to that location will retrieve the value.

**curl http://localhost:10018/riak/pgganimals/MCvHmnPmhTGJMfp3F30oLaaGIuE**

DELETE will remove it.

**curl -i -X DELETE http://localhost:10018/riak/pgganimals/MCvHmnPmhTGJMfp3F30oLaaGIuE**

If we’ve forgotten any of our keys in a bucket, we can get them all with keys=true.

**curl** [**http://localhost:10018/riak/pgganimals?keys=true**](http://localhost:10018/riak/pgganimals?keys=true)

1. Links and link walking

We will put the dogs in cages

**curl -X PUT** [**http://localhost:10018/riak/pggcages/1**](http://localhost:10018/riak/pggcages/1) **\  
 -H "Content-Type: application/json" \  
 -H "Link: </riak/pgganimals/polly>; riaktag=\"contains\"" \  
 -d '{"room" : 101}'**

It returns a multipart/mixed dump of headers plus bodies of all linked keys/values.

It’s also a headache to look at, we’ll dig a bit more into this syntax.

If you’re not familiar with reading the multipart/mixed MIME type, the Content-Type definition describes a boundary string, which denotes the beginning and end of some HTTP header and body data.

--BcOdSWMLuhkisryp0GidDLqeA64  
some HTTP header and body data  
--BcOdSWMLuhkisryp0GidDLqeA64--

In our case, the data is what cage 1 links to: Polly Purebred.

When link walking, we can replace the underscores in the link spec to filter only values we want.

Cage 2 has two links, so performing a link spec request will return both the animal Ace contained in the cage and the cage 1 next\_to it.

To specify only following the pgganimals bucket, replace the first underscore with the bucket name.

**curl** [**http://localhost:10018/riak/pggcages/2/pgganimals,\_,\_**](http://localhost:10018/riak/pggcages/2/pgganimals,_,_)

Note that this link relationship is one-directional.

In effect, the cage we’ve just created knows that Polly is inside it, but no changes have been made to Polly.

We can confirm this by pulling up Polly’s data and checking that there have been no changes to the Link headers.

**curl -i** [**http://localhost:10018/riak/pgganimals/polly**](http://localhost:10018/riak/pgganimals/polly)

You can have as many metadata Links as necessary, separated by commas.

We’ll put Ace in cage 2 and also point to cage 1 tagged with *next\_to* so we know that it’s nearby.

**curl -X PUT** [**http://localhost:10018/riak/pggcages/2**](http://localhost:10018/riak/pggcages/2) **\  
-H "Content-Type: application/json" \  
-H "Link:</riak/pgganimals/ace>;riaktag=\"contains\",**</riak/pggcages/1>;riaktag=\"next\_to\"" \  
-d '{"room" : 101}'

What makes Links special in Riak is *link walking* (and a more powerful variant, linked mapreduce queries, which we investigate later)

Getting the linked data is achieved by appending a *link spec* to the URL that is structured like this: /\_,\_,\_.

The underscores (\_) in the URL represent wildcards to each of the link criteria: bucket, tag, keep.

First let’s retrieve all links from cage 1.

**curl** [**http://localhost:10018/riak/pggcages/1/\_,\_,\_**](http://localhost:10018/riak/pggcages/1/_,_,_)

It returns a multipart/mixed dump of headers plus bodies of all linked keys/values.

It’s also a headache to look at, we’ll dig a bit more into this syntax.

If you’re not familiar with reading the multipart/mixed MIME type, the Content-Type definition describes a boundary string, which denotes the beginning and end of some HTTP header and body data.

--BcOdSWMLuhkisryp0GidDLqeA64  
some HTTP header and body data  
--BcOdSWMLuhkisryp0GidDLqeA64--

In our case, the data is what cage 1 links to: Polly Purebred.

When link walking, we can replace the underscores in the link spec to filter only values we want.

Cage 2 has two links, so performing a link spec request will return both the animal Ace contained in the cage and the cage 1 next\_to it.

To specify only following the pgganimals bucket, replace the first underscore with the bucket name.

**curl** [**http://localhost:10018/riak/pggcages/2/pgganimals,\_,\_**](http://localhost:10018/riak/pggcages/2/pgganimals,_,_)

Or follow the cages *next to* this one by populating the tag criteria.

**curl** [**http://localhost:10018/riak/pggcages/2/\_,next\_to,\_**](http://localhost:10018/riak/pggcages/2/_,next_to,_)

The final underscore—keep—accepts a 1 or 0. keep is useful when following second-order links, or links following other links, which you can do by just appending another link spec.

Let’s follow the keys next\_to cage 2, which will return cage 1.

Next, we walk to the animals linked to cage 1.

Since we set keep to 0, Riak will not return the intermediate step (the cage 1 data).

It will return only Polly’s information, who is next to Ace’s cage.

**curl** [**http://localhost:10018/riak/pggcages/2/\_,next\_to,0/pgganimals,\_,\_**](http://localhost:10018/riak/pggcages/2/_,next_to,0/pgganimals,_,_)

If we want Polly’s information and cage 1, set keep to 1.

**curl** [**http://localhost:10018/riak/pggcages/2/\_,next\_to,1/\_,\_,\_**](http://localhost:10018/riak/pggcages/2/_,next_to,1/_,_,_)

1. Beyond Links

Along with Links, you can store arbitrary metadata by using the X-Riak-Meta- header prefix.

If we wanted to keep track of the color of a cage but it wasn’t necessarily important in the day-to-day cage-managing tasks at hand, we could mark cage 1 as having the color pink.

Getting the URL’s header (the -I flag) will return your metadata name and value.

**curl -X PUT** [**http://localhost:10018/riak/pggcages/1**](http://localhost:10018/riak/pggcages/1) **\  
 -H "Content-Type: application/json" \  
 -H "X-Riak-Meta-Color: Pink" \  
 -H "Link: </riak/pgganimals/polly>; riaktag=\"contains\"" \  
 -d '{"room" : 101}'**

1. Mime Types

Copy the image called polly.jpg from the home directory to your directory ( move to the /home directory and issue the command cp polly.jpg /home/Tnumber)

**curl -X PUT** [**http://localhost:10018/riak/pggphotos/polly.jpg**](http://localhost:10018/riak/pggphotos/polly.jpg) **\  
 -H "Content-type: image/jpeg" \  
 -H "Link: </riak/pgganimals/polly>; riaktag=\"photo\"" \  
 --data-binary @polly.jpg**

Now visit the URL in a web browser, which will be delivered and rendered exactly as you’d expect any web client-server request to function (can’t do this today since we are using only the command line – instead you can get the image using curl

curl <http://localhost:10018/riak/pggphotos/polly.jpg>

1. Nodes/Writes/Reads

Riak allows us to control reads and writes into the cluster by altering three values: N, W, and R. *N* is the number of nodes a write ultimately replicates to, in other words, the number of copies in the cluster.

*W* is the number of nodes that must be successfully written to before a successful response.

If W is less than N, a write will be considered successful even while Riak is still copying the value.

Finally, *R* is the number of nodes required to read a value successfully.

If R is greater than the number of copies available, the request will fail.

Let’s investigate each of these in more detail.

When we write an object in Riak, we have the choice to replicate that value across multiple nodes.

The benefit here is that if one server goes down, then a copy is available on another.

The n\_val bucket property stores the number of nodes to replicate a value to (the N value); it’s 3 by default.

We can alter a bucket’s properties by putting a new value in the props object.

Here we set animals to have an n\_val of 4:

Note use your initials in front of the bucket name

**curl -X PUT** [**http://localhost:10018/riak/pgganimals**](http://localhost:10018/riak/pgganimals) **\  
 -H "Content-Type: application/json" \  
 -d '{"props":{"n\_val":4}}'**

*N* is simply the total number of nodes that will *eventually* contain the correct value.

This doesn’t mean we must wait for the value to replicate to *all* of those nodes in order to return.

Sometimes we just want our client to return immediately and let Riak replicate in the background.

Or sometimes we want to wait until Riak has replicated to all *N* nodes (just to be safe) before returning.

We can set the *W* value to the number of successful writes that must occur before our operation is considered a success.

Although we’re writing to four nodes eventually, if we set W to 2, a write operation will return after only two copies are made.

The remaining two will replicate in the background

curl -X PUT <http://localhost:10018/riak/pgganimals> \  
 -H "Content-Type: application/json" \  
 -d '{"props":{"w":2}}'

Finally, we can use the *R* value.

*R* is the number of nodes that must be read in order to be considered a successful read.

You can set a default *R* like we did earlier with n\_val and w.

curl -X PUT <http://localhost:10018/riak/pgganimals> \  
 -H "Content-Type: application/json" \  
 -d '{"props":{"r":3}}'

But Riak provides a more flexible solution.

We may choose the number of nodes we want to read by setting an r parameter in the URL *per request*.

curl <http://localhost:10018/riak/pgganimals/ace?r=3>

You may be asking yourself why we would ever need to read from more than one node.

After all, values we write will eventually be replicated to *N* nodes, and we can read from any of them.

We find the idea is easier to visualize.

Let’s say we set our NRW values to {"n\_val":3, "r":2, "w":1}, like Figure 9, *Eventual consistency: W+R <= N*, on page 76.

This makes our system more responsive on writes, since only one node needs to be written before returning.

But there is a chance that another operation could perform a read before the nodes had a chance to synchronize.

Even if we read from two nodes, it’s possible we could receive an old value.

One way to be certain we have the most current value is to set W=N and R=1 like this: {"n\_val":3, "r":1, "w":3} (see Figure 10, *Consistency by writes: W=N, R=1*, on page 76).

In essence, this is what relational databases do; they enforce consistency by ensuring a write is complete before returning.

We can certainly read faster, since we need to access only one node.

But this can really slow down writes.

Or you could just write to a single node but read from all of them.

This would be setting W=1 and R=N like this: {"n\_val":3, "r":3, "w":1} (see Figure 11, *Consistency by reads: W=1, R=N*, on page 76).

Although you may read a few old values, you are guaranteed to retrieve the most recent value, too.

You’ll just have to resolve which one that is (using a vector clock, which we’ll cover later).

Of course, this has the opposite problem as shown earlier and slows down reads.

Or you could set W=2 and R=2 as {"n\_val":3, "r":2, "w":2} (see Figure 12, *Consistency by quorum: W+R > N*, on page 77).

This way, you need only write to more than half of the nodes and read from more than half, but you still get the benefits of consistency while sharing the time delays between reads and writes.

This is called a *quorum* and is the minimum amount to keep consistent data.

You are free to set your R or W to any values between 1 and N but will generally want to stick with one, all, or a quorum.

These are such common values that R and W can accept string values representing them, defined in the following table:

**One** This is just the value 1. Setting W or R means only one node need respond for the request to succeed.

**All** This is the same value as N. Setting W or R to this means all replicated nodes must respond.

**Quorum** This equals setting the value to N/2+1. Setting W or R means most nodes must respond to succeed.

**Default** Whatever the W or R value is set for the bucket. Generally defaults to 3.

In addition to the previous values as valid bucket properties, you can also use them as query parameter values.

curl <http://localhost:10018/riak/pgganimals/ace?r=all>

11 Vector Clocks

For this example we want to see all conflicting versions so we can resolve them manually.

Let’s keep multiple versions by setting the allow\_mult property on the animals bucket.

Any key with multiple values are called *sibling* values.

$ curl -X PUT <http://localhost:10018/riak/pgganimals> \  
 -H "Content-Type: application/json" \  
 -d '{"props":{"allow\_mult":true}}'

Here, Bob puts Bruiser in the system with his chosen score of 3 and a client ID of *bob*.

$ curl -i -X PUT <http://localhost:10018/riak/pgganimals/bruiser> \  
 -H "X-Riak-ClientId: bob" \  
 -H "Content-Type: application/json" \  
 -d '{"score" : 3}'

Jane and Rakshith both pull Bruiser’s data that Bob created (you’ll have much more header information; we’re just showing the vector clock here).

Note that Riak encoded Bob’s vclock, but under the covers it’s a client and a version (and timestamp, so yours will be different from the one shown).

$ curl -i <http://localhost:10018/riak/pgganimals/bruiser?return_body=true>

Jane makes her update to score 2 and includes the most recent vector clock she received from Bob’s version.

This is a signal to Riak that her value is an update of Bob’s version.

$ curl -i -X PUT <http://localhost:10018/riak/pgganimals/bruiser> \  
 -H "X-Riak-ClientId: jane" \  
 -H "X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==" \  
 -H "Content-Type: application/json" \  
 -d '{"score" : 2}'

Since Jane and Rakshith pulled Bob’s data at the same time, he also submits an update (of score 4) using Bob’s vector clock.

curl -i -X PUT <http://localhost:10018/riak/pgganimals/bruiser> \  
 -H "X-Riak-ClientId: rakshith" \  
 -H "X-Riak-Vclock: a85hYGBgzGDKBVI8R4M2cltNMbFnYJjjlMGUyJjHyhD0IPU8XxYA" \  
 -H "Content-Type: application/json" \  
 -d '{"score" : 4}'

When Jane rechecks the score, she sees not a value, as expected, but rather an HTTP code for multiple choices and a body containing two “sibling” values.

$ curl <http://localhost:10018/riak/pgganimals/bruiser?return_body=true>

Siblings:  
637aZSiky628lx1YrstzH5  
7F85FBAIW8eiD9ubsBAeVk

Riak stored these versions in a multipart format, so she can retrieve the entire object by accepting that MIME type.

curl -i <http://localhost:10018/riak/pgganimals/bruiser?return_body=true> \  
 -H "accept: multipart/mixed"

Jane’s job now is to use this information to make a reasonable update.

She decides to average the two scores and update to 3, using the vector clock given to resolve the conflict.

$ curl -i -X PUT <http://localhost:10018/riak/pgganimals/bruiser?return_body=true> \  
-H "X-Riak-ClientId: jane" \  
-H "X-Riak-Vclock: a85hYGBgymDKBVIs7NtEXmUwJTLmsTI8FMs5zgcR5jkatJHbaoqJPQPDHCegNDNQ+t6j1PN8WQA=" \  
-H "Content-Type: application/json" \

-d '{"score" : 3}'

Now when Bob and Rakshith retrieve bruiser’s information, they’ll get the resolved score.

$ curl -i <http://localhost:10018/riak/pgganimals/bruiser?return_body=true>

HTTP/1.1 200 OK  
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...CpQmAkonCcHFM4CAA==

{"score" : 3}

Any future requests will receive score 3.