Riak

Riak is a distributed key-value database where values can be anything—from plain text, JSON, or XML to images or video clips—all accessible through a simple HTTP interface.

Riak is also fault-tolerant. Servers can go up or down at any moment with no single point of failure.

 But this flexibility has some trade-offs.

Riak lacks robust support for ad hoc queries, and key-value stores, by design, have trouble linking values together (in other words, they have no foreign keys).

Riak Loves the Web

You query via URLs, headers, and verbs, and Riak returns assets and standard HTTP response codes.

Riak is a great choice for datacenters like Amazon that must serve many requests with low latency.

It’s easy to manage, easy to set up, and can grow with your needs.

If you’ve ever used Amazon Web Services, like SimpleDB or S3, you may notice some similarities in form and function. This is no coincidence. Riak is inspired by Amazon’s Dynamo paper.

First, we’ll investigate how Riak stores and retrieves values and how to tie data together using Links.

Then we’ll explore mapreduce.

We’ll see how Riak clusters its servers and handles requests, even in the face of server failure.

Finally, we’ll look at how Riak resolves conflict that arises from writing to distributed servers, and we’ll look at some extensions to the basic server.

CRUD, Links, and MIMEs

You can download and install a build of Riak provided by Basho the company that funds its development. Erlang3 is also required to run Riak (R16B02 or greater).

We will use a cloud hosted instance of Riak.

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

cd /home/Tnumber

Type pwd – you should be in the folder /home/Tumber, if not then move to your home folder

Riak supplies an HTTP REST interface, so we’re going to interact with it via the URL tool cURL. In production, you’ll almost always use a driver in your language of choice. Using cURL allows us to peek at the underlying API without resorting to a particular driver or programming language.

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

On the machine you install Riak you will find three example servers. Just fire them up:

(or from the dev directory

for node in dev\*; do $node/bin/riak start; done)

**$ dev/dev1/bin/sudo ./riak start  
$ dev/dev2/bin/sudo ./riak start  
$ dev/dev3/bin/sudo ./riak start**

We should now have three processes representing individual Riak nodes (server instances), unaware of each other’s presence.

*(-- done*

*To create a cluster, we need to join the nodes using each server’s riak-admin command named join and point them to any other cluster node.*

***$ dev/dev2/bin/riak-admin join –f dev1@127.0.0.1***

*It doesn’t really matter which servers we point them at—in Riak, all nodes are equal.*

*Now that dev1 and dev2 are in a cluster, we can point dev3 at either one.*

***$ dev/dev3/bin/riak-admin join –f*** [***dev2@127.0.0.1***](mailto:dev2@127.0.0.1)

*--Done )*

Verify your servers are healthy by checking their stats using

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

It may prompt you to download the file, which contains lots of information about the cluster. 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}]"*, ...

We can see that all servers are equal participants in the ring by pinging the other servers for stats on ports 10018 (dev1),10028 (dev2) and 10038 (dev3).

For now, we’ll stick with the stats from dev1.

Look for the *ring\_members* property—it should contain all our node names and will be the same for each server.

Next, find the value for *connected\_nodes*. This should be a list of the other servers in the ring.

We can change the values reported by *connected\_nodes* by stopping a node...

**dev/dev2/bin/riak stop**

...and reloading the /stats.

Notice that dev2@127.0.0.1 is now gone from the *connected\_nodes* list.

Start dev2, and it will rejoin itself to the *Riak ring*

REST Interface

REST stands for REpresentational State Transfer. It has become the *de facto* architecture of web applications, so it’s worth knowing.

REST is a guideline for mapping resources to URLs and interacting with them using CRUD verbs: POST (Create), GET (Read), PUT (Update), and DELETE (Delete).

We use Curl as our REST interface, because it’s easy to specify verbs (like GET and PUT) and HTTP header information (like Content-Type).

With the curl command, we speak directly to the Riak server’s HTTP REST interface without the need for an interactive console or, say, a Ruby or Java driver.

You can validate the curl command works with Riak by pinging a node. **$ curl** [**http://localhost:10018/ping**](http://localhost:8091/ping)

OK

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

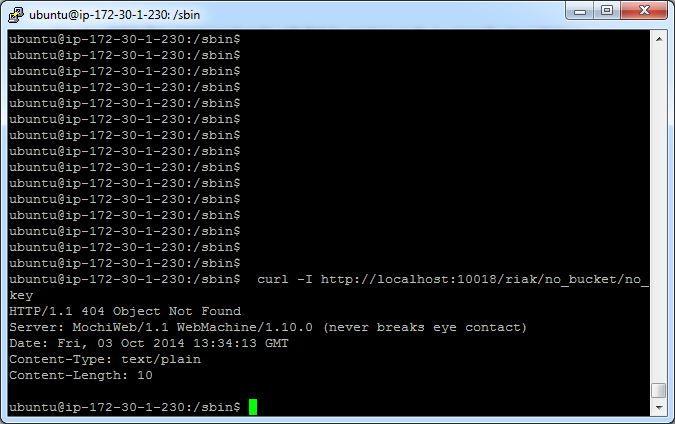
**curl -I** [**http://localhost:10018/riak/no\_bucket/no\_key**](http://localhost:8091/riak/no_bucket/no_key)

HTTP/1.1 404 Object Not Found

Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic) Date: Thu, 04 Aug 2011 01:25:49 GMT

Content-Type: text/plain

Content-Length: 10



Since Riak leverages HTTP URLs and actions, it uses HTTP headers and error codes.

The 404 response means the same as a 404 when you encounter a missing web page: nothing to see here.

So, let’s PUT something in Riak.

The -X PUT parameter tells cURL that we want to perform an HTTP PUT action to store and retrieve on an explicit key.

The -H attribute sets the following text as HTTP header information.

In this case, we set the MIME content type to HTML.

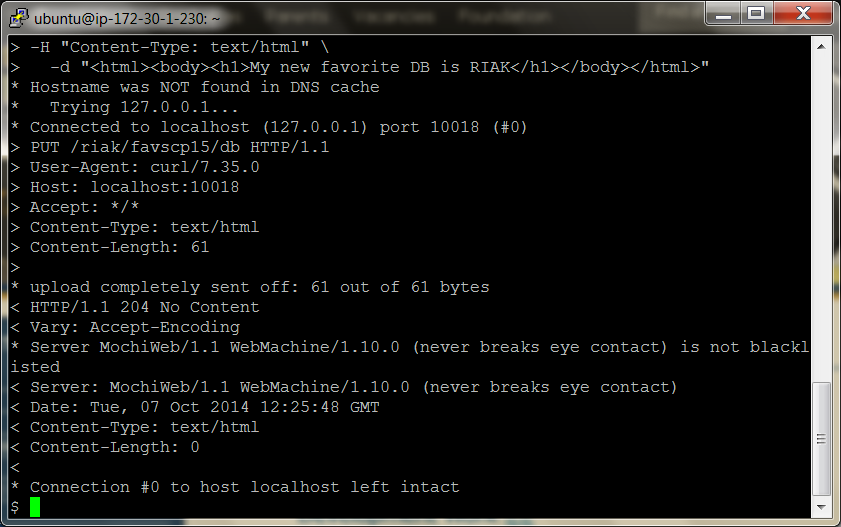
In order that each of us has their own database please put your initials before each bucket you create

Everything passed to -d (also known as the body data) is what Riak will add as a new value.

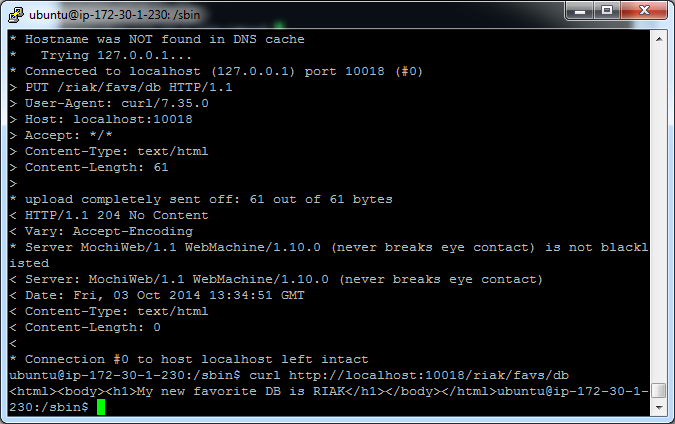
In the examples below use your initials in the bucket names

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> in a browser, you’ll get a nice message from yourself.



PUT the Value in the Bucket

Riak is a key-value store, so it expects you to pass in a key to retrieve a value.

Riak breaks up classes of keys into *buckets* to avoid key collisions—for example, a key for java the *language* will not collide with java the *drink*.

We’re going to create a system to keep track of animals in a dog hotel.

We’ll start by creating a bucket of animals

The URL follows this pattern:

<http://SERVER:PORT/riak/BUCKET/KEY>

A straightforward way of populating a Riak bucket is to know your key in advance.

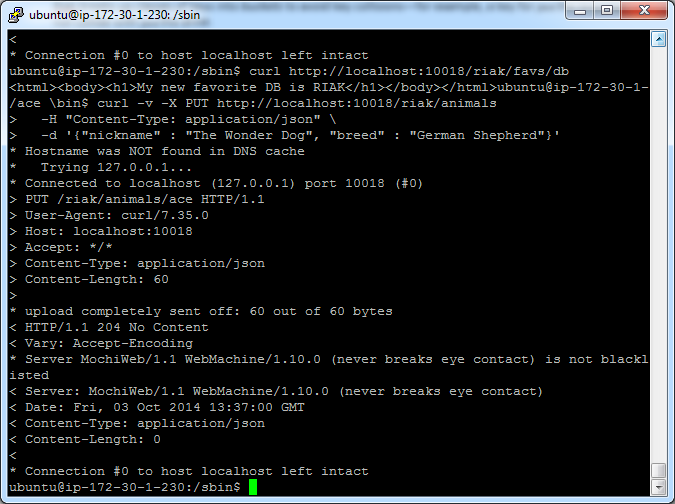
We’ll first add *Ace, The Wonder Dog* and give him the key ace with the value {"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}.

You don’t need to explicitly create a bucket—putting a first value into a bucket name will create that bucket.

Remember to use /riak/<your initials>animals/ace below

**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"}'**

Putting a new value returns a 204 code.



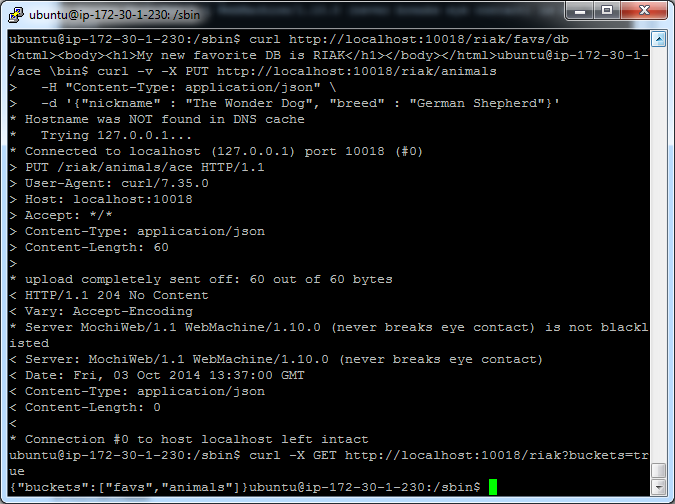
The -v (verbose) attribute in the curl command outputs this header line.

< HTTP/1.1 204 No Content

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

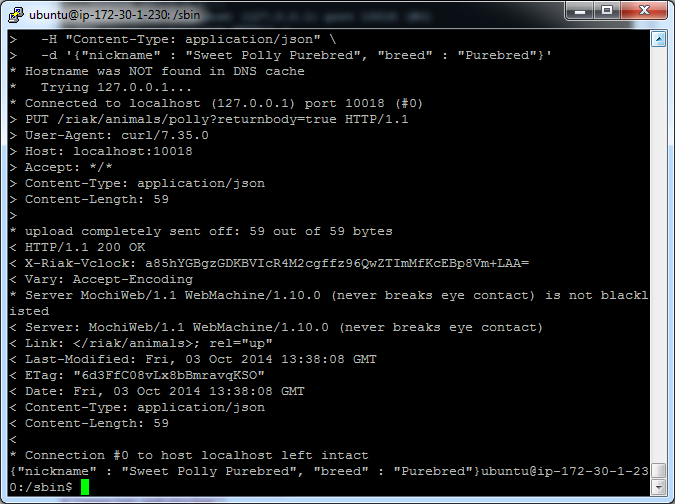
**curl -X GET** [**http://localhost:10018/riak?buckets=true**](http://localhost:8091/riak?buckets=true)

{"buckets":["pggfavs","pgganimals"]}



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"}'**

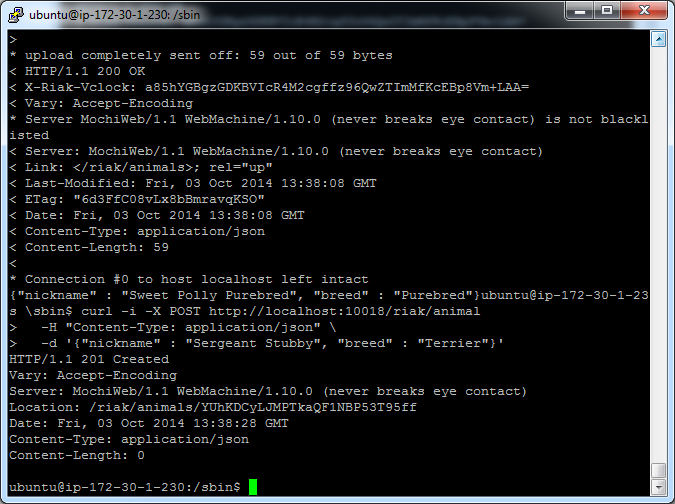


This time you’ll see a 200 code.

< HTTP/1.1 200 OK

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"}'**

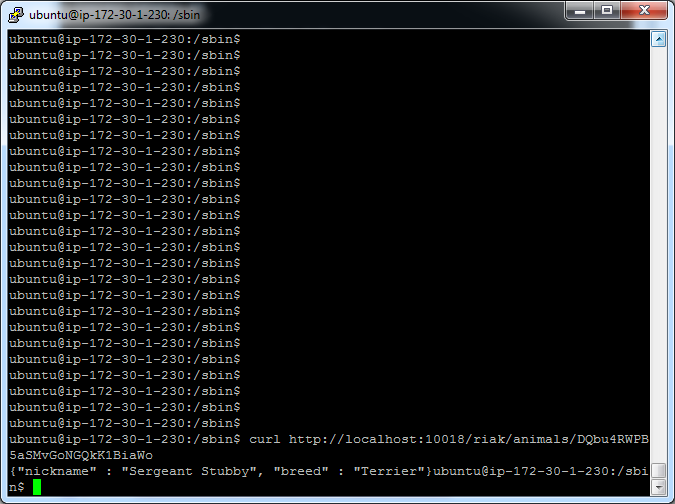


The generated key will be in the header under Location—also note the 201 success code in the header.

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**

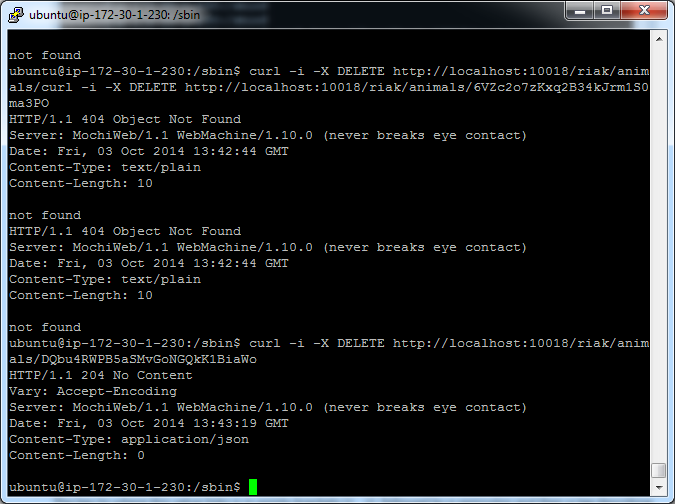
HTTP/1.1 204 No Content

Vary: Accept-Encoding

Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)

Date: Mon, 11 Apr 2011 05:08:39 GMT  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 0

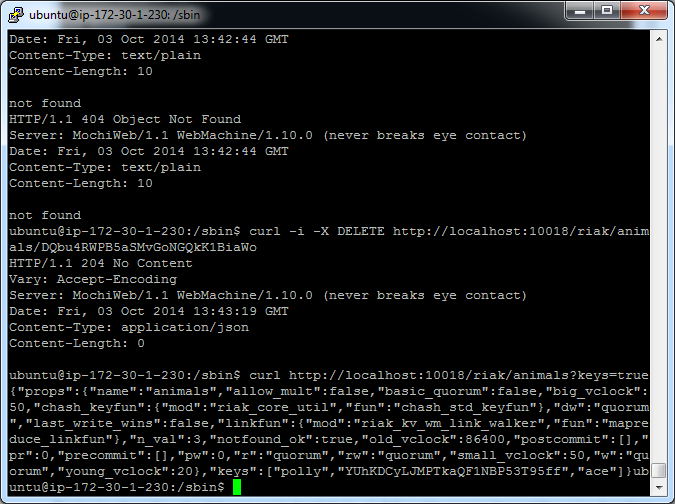
DELETE won’t return any body, but the HTTP code will be 204 if successful.



Otherwise, as you’d expect, it returns a 404.

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)



You can also get them as a stream with keys=stream, which can be a safer choice for huge datasets—it just keeps sending chunks of keys array objects and ends with an empty array.

Links

Links are metadata that associate one key to other keys.

The basic structure is this:

Link: </riak/bucket/key>; riaktag=\"whatever\"

The key to where this value links is in pointy brackets (<...>), followed by a semicolon and then a tag describing how the link relates to this value (it can be whatever string we like).

Link Walking

Our little dog hotel has quite a few cages.

To keep track of which animal is in what cage, we’ll use a link.

Cage 1 contains Polly by linking to her key (this also creates a new bucket named pgcages).

The cage is installed in room 101, so we set that value as JSON data.

Remember to use /cp1cages/ and cp1animals/ in the example below

**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}'**

Note that this link relationship is one-directional.

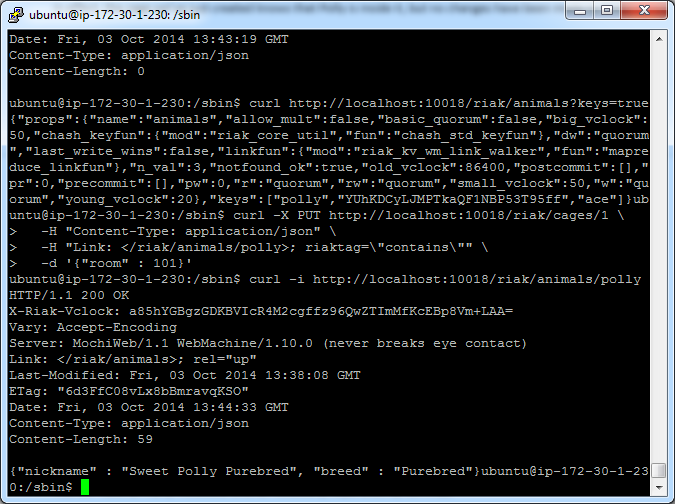
In effect, the cage we’ve just created knows that Polly is inside it, but no changes have been made to Polly.

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)

HTTP/1.1 200 OK  
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=  
Vary: Accept-Encoding  
Server: MochiWeb/1.1 WebMachine/1.9.0 (participate in the frantic)  
Link: </riak/pgganimals>; rel="up"  
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT  
ETag: "VD0ZAfOTsIHsgG5PM3YZW"  
Date: Tue, 13 Dec 2011 17:54:51 GMT  
Content-Type: application/json  
Content-Length: 59

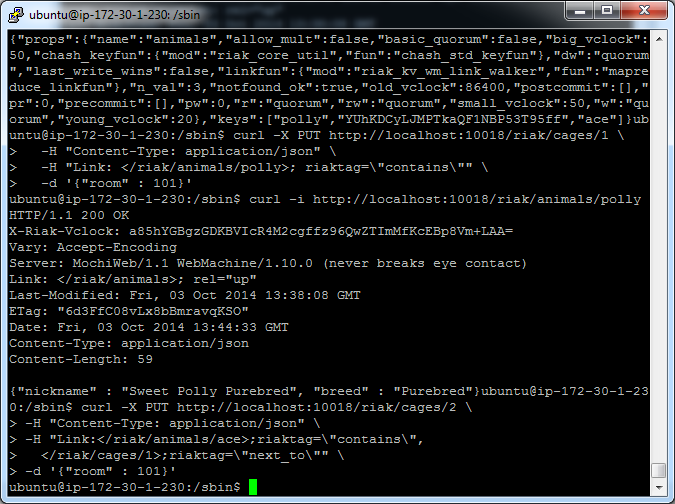
{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}



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/_,_,_)

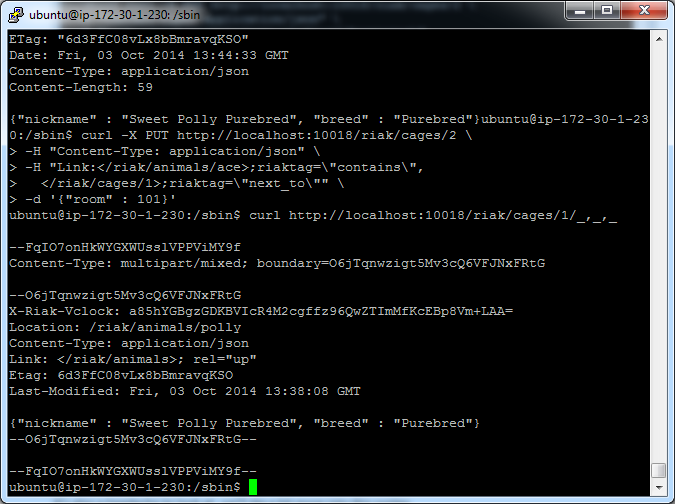
--4PYi9DW8iJK5aCvQQrrP7mh7jZs

Content-Type: multipart/mixed; boundary=Av1fawIA4WjypRlz5gHJtrRqklD

--Av1fawIA4WjypRlz5gHJtrRqklD  
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=  
Location: /riak/pgganimals/polly  
Content-Type: application/json  
Link: </riak/pgganimals>; rel="up"  
Etag: VD0ZAfOTsIHsgG5PM3YZW  
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}  
--Av1fawIA4WjypRlz5gHJtrRqklD--

--4PYi9DW8iJK5aCvQQrrP7mh7jZs--



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--

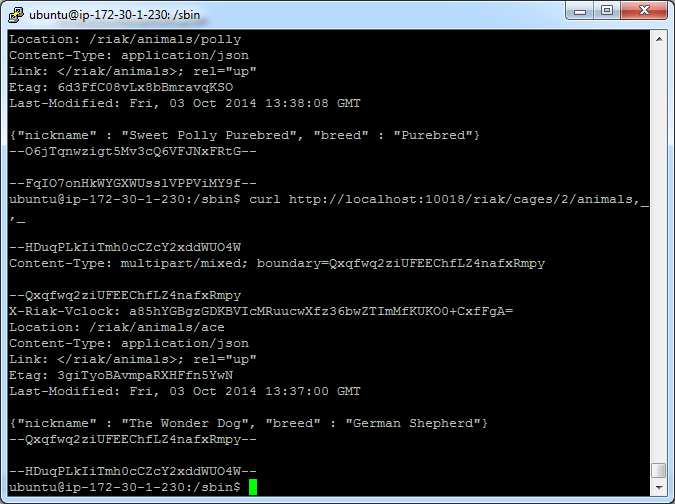
In our case, the data is what cage 1 links to: Polly Purebred.

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.

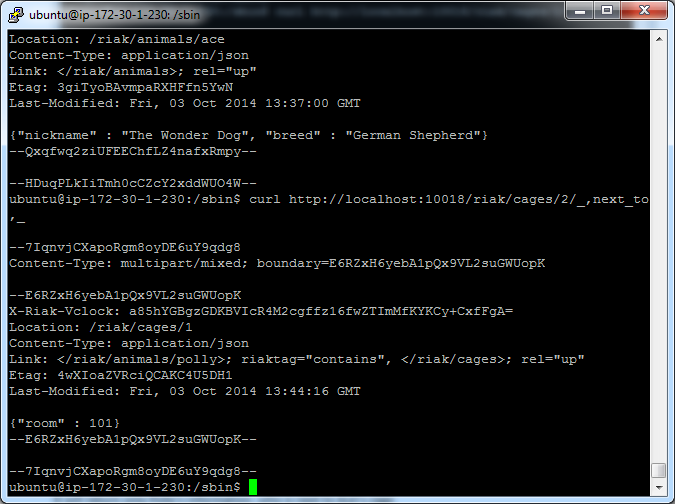
To specify only following the pgganimals bucket, replace the first underscore with the bucket name.

**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,_,_)

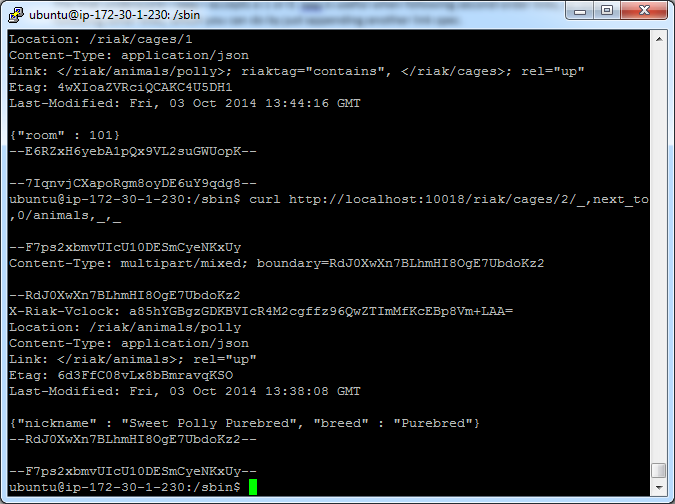
--6mBdsboQ8kTT6MlUHg0rgvbLhzd

Content-Type: multipart/mixed; boundary=EZYdVz9Ox4xzR4jx1I2ugUFFiZh

--EZYdVz9Ox4xzR4jx1I2ugUFFiZh  
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=  
Location: /riak/animals/polly  
Content-Type: application/json  
Link: </riak/animals>; rel="up"  
Etag: VD0ZAfOTsIHsgG5PM3YZW  
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

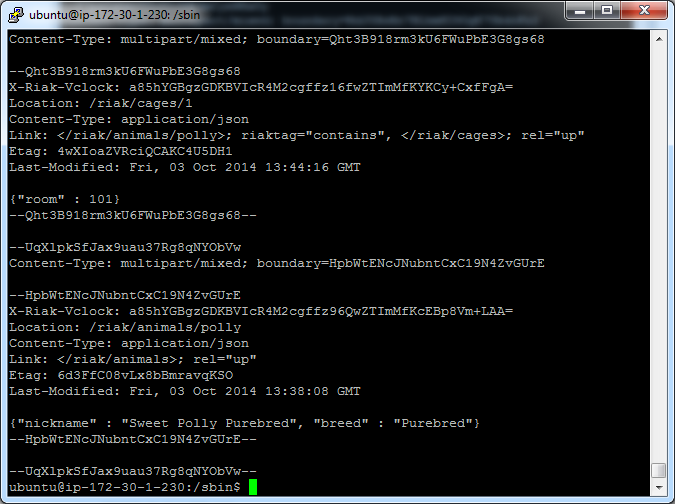
{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}  
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh--

--6mBdsboQ8kTT6MlUHg0rgvbLhzd--



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

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



--PDVOEl7Rh1AP90jGln1mhz7x8r9  
Content-Type: multipart/mixed; boundary=YliPQ9LPNEoAnDeAMiRkAjCbmed

--YliPQ9LPNEoAnDeAMiRkAjCbmed  
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRKY+VIYo35gRfFgA=

Location: /riak/cages/1  
Content-Type: application/json  
Link: </riak/animals/polly>; riaktag="contains", </riak/cages>; rel="up"  
Etag: 6LYhRnMRrGIqsTmpE55PaU  
Last-Modified: Tue, 13 Dec 2011 17:54:34 GMT

{"room" : 101}

--YliPQ9LPNEoAnDeAMiRkAjCbmed--

--PDVOEl7Rh1AP90jGln1mhz7x8r9  
Content-Type: multipart/mixed;

boundary=GS9J6KQLsI8zzMxJluDITfwiUKA

--GS9J6KQLsI8zzMxJluDITfwiUKA  
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=  
Location: /riak/animals/polly  
Content-Type: application/json  
Link: </riak/animals>; rel="up"  
Etag: VD0ZAfOTsIHsgG5PM3YZW  
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}  
--GS9J6KQLsI8zzMxJluDITfwiUKA--

--PDVOEl7Rh1AP90jGln1mhz7x8r9--

This returns the objects in the path to the final result. In other words, *keep* the step.

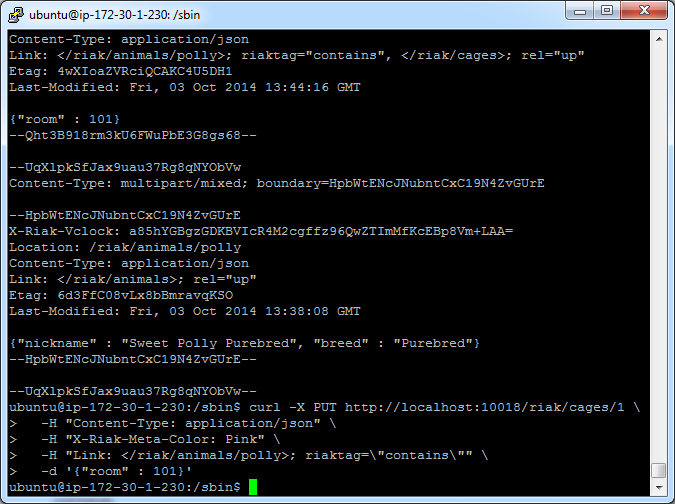
Beyond Links

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}'**



MIME Types in Riak

Riak stores everything as a binary-encoded value, just like normal HTTP.

The MIME type gives the binary data context—we’ve been dealing only with plain text up until now.

MIME types are stored on the Riak server but are really just a flag to the client so that when it downloads the binary data, it knows how to render it.

We’d like our dog hotel to keep images of our guests.

We need only use the data-binary flag on the curl command to upload an image to the server and specify the MIME type as image/jpeg.

We’ll add a link back to the /pgganimals/polly key so we know who we are looking at.

First, create an image called polly.jpg and place it in the same directory you’ve been using to issue the curl commands (should already be in the /home 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>

Since we pointed the image to /pgganimals/polly, we could link walk from the image key *to* Polly but not vice versa.

You link the direction you need to walk.

If we believe our use case will require accessing image data from the animals bucket, a link should exist on that object instead (or in addition).

Mapreduce and Server Clusters

The mapreduce framework can perform more powerful queries than the standard key-value paradigm can normally provide.

Finally, we will investigate the server architecture of Riak and how it uses a novel server layout to provide flexibility in consistency or availability, even in the face of network partitions.

Population Script

*A quick populator script in Ruby (installed) will create data for a gigantic 10,000-room hotel.*

*STYLES = single double queen king suite*

*bucket = rooms*

*ro.data = {'style' => style, 'capacity' => capacity}*

*)*

Mapreduce

One of Google’s greatest lasting contributions to computer science is the popularization of mapreduce as an algorithmic framework for executing jobs in parallel over several nodes.

It is described in Google’s seminal paper on the topic and has become a valuable tool for executing custom queries in the class of partition-tolerant datastores.

Mapreduce breaks down problems into two parts.

Part 1 is to convert a list of data into another type of list by way of a map() function.

Part 2 is to convert this second list to one or more scalar values by way of a reduce() function.

Following this pattern allows a system to divide tasks into smaller components and run them across a massive cluster of servers in parallel.

 In one way, mapreduce is the opposite of how we normally run queries.

A Ruby system might grab data like this:

*# Construct a Hash to store room capacity count keyed by room style*

capacity\_by\_style = {}

rooms = Room.all

**for** room **in** rooms

total\_count = capacity\_by\_style[room.style]

capacity\_by\_style[room.style] = total\_count.to\_i + room.capacity

**end**

Room.all runs an SQL query against the backing database similar to SELECT \* FROM rooms;

The database sends all of the results to the application, and the application code performs some action on that data.

In this case, we’re looping through each room in the hotel and then counting the total capacity for each room style (for example, the capacity of all the suites in the hotel may be 448 guests).

This is acceptable for small datasets.

But as room count grows, the system slows as the database continues to stream each room’s data to the application.

Mapreduce runs in an inverse manner.

Rather than grabbing data from the database and running it on a client application, mapreduce is a pattern to pass an algorithm to all of the database nodes, which are then each responsible for returning a result.

*Each object on the server is “mapped” to some common key that groups the data together, and then all matching keys are “reduced” into some single value*

In Figure 7, *The map function outputs*, on page 65 we can see how a bucket of phone bills keyed by phone number may calculate the total charged against all numbers across three servers, where each server contains all numbers with a similar prefix.

For Riak, that means the database servers are responsible for mapping and reducing the values on each node.

Those reduced values are passed around, where some other server (usually the requesting server) reduces those values further, until a final result is passed to the requesting client ).

This simple reversal is a powerful way to allow complex algorithms to run locally on each server and return a very small result to the calling client.

*It’s faster to send the algorithm to the data and then send the data to the algorithm.*

Mapreduce in Riak

I will be using a standalone riak server on 8098 for the mapreduce exercises

Stop ring servers ./riak stop in each /bin folder

Start standalone server from rel/riak/bin

Let’s create mapreduce functions for our Riak dataset.

A neat feature of Riak’s mapreduce is that you can run the map() function alone and see what all the results are mid-run (assuming you even want to run a reduce).

Let’s look at the results for rooms 101, 102, and 103 only.

The map setting needs the language we’re using and the source code; only then do we actually write the JavaScript map function (the function is just a string, so we always need to escape any characters accordingly).

Using the @- command in cURL keeps the console’s standard input open until receiving CTRL+D.

This data will populate the HTTP body sent to the URL, which we post to the /mapred command (look carefully—the URL is /mapred, not /riak/mapred).

 In the next commands paste the first line and hit enter, then the code, then finally ctrl-d

**curl -X POST -H "content-type:application/json"** [**http://localhost:8098/mapred**](http://localhost:8098/mapred) **--data @-**

{

"inputs":[  
 ["rooms","101"],["rooms","102"],["rooms","103"],["rooms","104"],["rooms","105"]

], "query":[

{"map":{  
 "language":"javascript",  
 "source":

"function(v) {

/\* From the Riak object, pull data and parse it as JSON \*/

var parsed\_data = JSON.parse(v.values[0].data);

var data = {};

/\* Key capacity number by room style string \*/

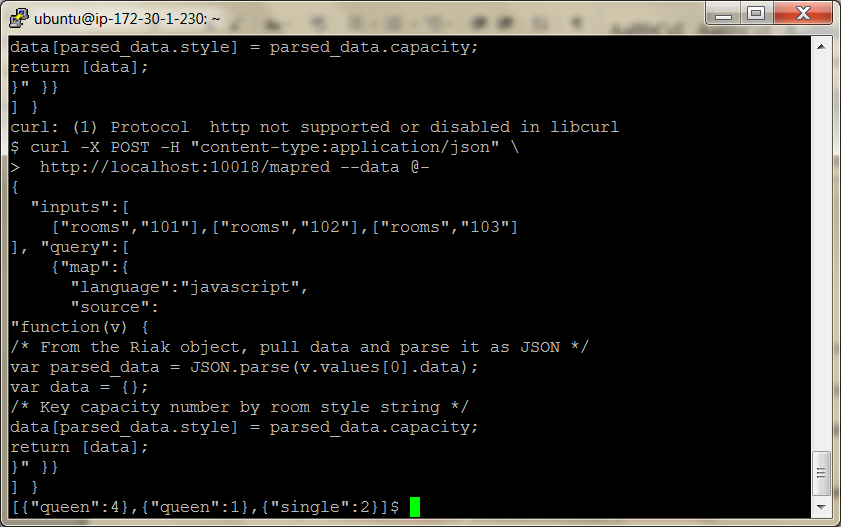
**data[parsed\_data.style] = parsed\_data.capacity;**

return [data];

}" }}

] }

CTRL-D



The /mapred command expects valid JSON, and here we specified the form of our mapreduce commands.

We choose the three rooms we want by setting the “inputs” value to be an array containing [bucket, key] pairs.

But the real meat of the settings is under the *query* value, which accepts an array of JSON objects containing objects, keyed by *map*, *reduce*, and/or *links* (more on links later).

All this does is dig down into the data (v.values[0].data), parse the value as a JSON object (JSON.parse(...)), and return the capacity (parsed\_data.capacity) keyed by room style (parsed\_data.style).

You’ll get a result like this:

[{"suite":6},{"single":1},{"double":1}]

It’s just the three objects’ JSON data from rooms 101, 102, and 103.

We didn’t need to simply output the data as JSON.

We could have converted the value of each key value into anything we wanted.

We dug into the body data only but could have retrieved metadata, link information, the key, or data.

Anything is possible after that—we are mapping each key value into some other value.

If you feel up to it, you can return the maps of all 10,000 rooms by replacing the input-specific [bucket, key] arrays with the rooms bucket name, like this:

"inputs":"rooms"

Fair warning: it will dump a lot of data.

Stored Functions

Another option Riak provides us with is to store the map function in a bucket value.

This is another example of moving the algorithm to the database.

This is a stored procedure or, more specifically, a user-defined function—of similar philosophy to those used in relational databases for years.

Remember to use <your initials>my\_functions/

**curl -X PUT -H "content-type:application/json"** [**http://localhost:8098/riak/pggmy\_functions/map\_capacity**](http://localhost:8098/riak/pggmy_functions/map_capacity) **--data @-**

function(v) {

var parsed\_data = JSON.parse(v.values[0].data);

var data = {};

**data[parsed\_data.style] = parsed\_data.capacity;**

return [data];

}

Ctrl D

With your function safely stored, we’ll run the function by pointing to the new bucket and key containing the function.

**curl -X POST -H "content-type:application/json"** [**http://localhost:8098/mapred**](http://localhost:8098/mapred) **--data @-**

{

"inputs":[  
 ["rooms","101"],["rooms","102"],["rooms","103"]

], "query":[

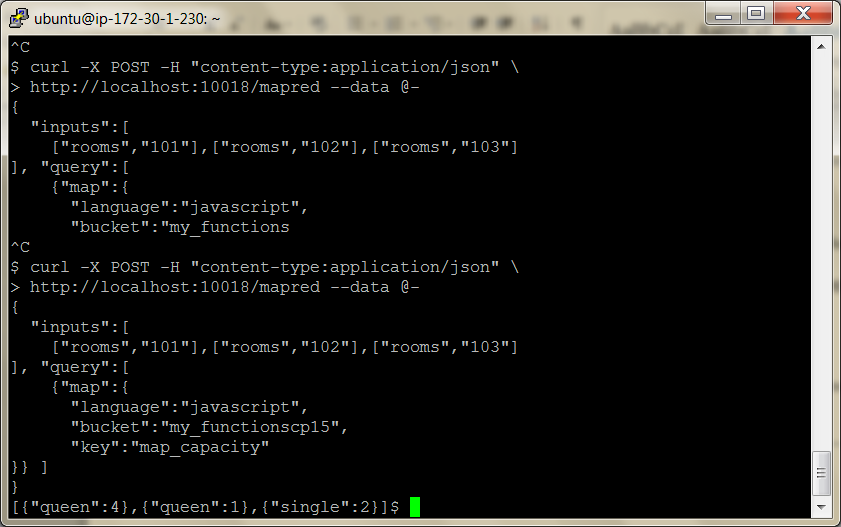
{"map":{  
 "language":"javascript",  
 "bucket":"pggmy\_functions",  
 "key":"map\_capacity"

}} ]

}

Ctrl D

You should receive the same results you received by putting the JavaScript source inline.



Reducing

Mapping is useful, but you’re limited to converting individual values into other individual values.

Performing some sort of analysis over that set of data, even something as simple as counting the records, requires another step.

This is where reduce comes into play.

The SQL/Ruby example that we looked at earlier showed how each value could be iterated over and how capacity was totaled for each style of room.

We will perform this in our reduce function in JavaScript.

Most of the command we pass to /mapred will be the same.

This time, we add the reduce function.

**curl -X POST -H "content-type:application/json"** [**http://localhost:8098/mapred**](http://localhost:8098/mapred) **--data @-**

{

"inputs":"rooms",  
 "query":[

{"map":{  
 "language":"javascript",  
 "bucket":"pggmy\_functions",  
 "key":"map\_capacity"

}},  
 {"reduce":{

"language":"javascript",  
 "source":

"function(v) {  
 var totals = {};  
 for (var i in v) {

for(var style in v[i]) {  
 if( totals[style] )

totals[style] += v[i][style];

else

totals[style] = v[i][style];

} }

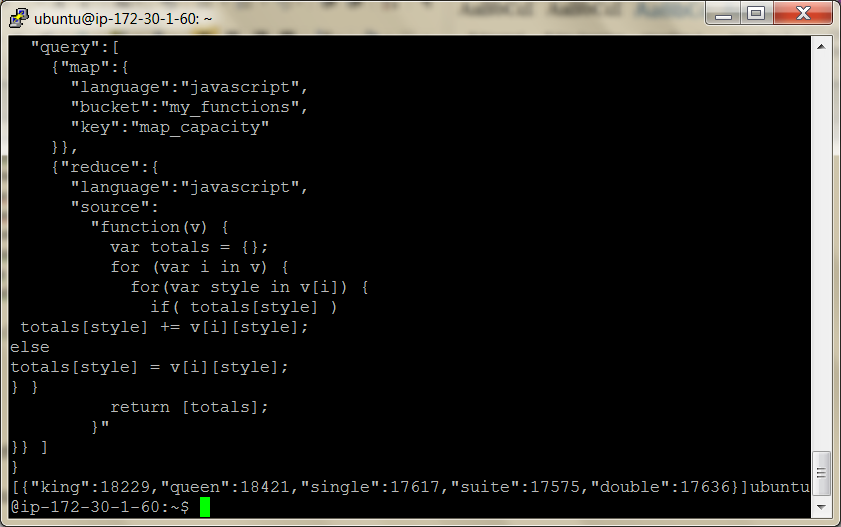
return [totals];  
 }"

}} ]

}

Ctrl D

Running this on all rooms should return total counts of capacity, keyed by room style.



Your totals won’t match the previous exactly, since we randomly generated room data

Consistency and Durability

Riak server architecture removes single points of failure (all nodes are peers) and allows you to grow or shrink the cluster at will.

This is important when dealing with large-scale deployments, since it allows your database to remain available even if several nodes fail or are otherwise unresponsive.

Distributing data across several servers is saddled with an inherent problem.

If you want your database to continue running when a network partition occurs (meaning, some messages were lost), it means you must make a trade- off.

Either you can remain *available* to server requests or you can refuse requests and ensure the *consistency* of your data.

It is not possible to create a distributed database that is fully consistent, available, and partition tolerant.

You can have only two (partition tolerant and consistent, partition tolerant and available, or consistent and available but not distributed). This is known as the CAP theorem (Consistency, Availability, Partition tolerance).

It is a problem in system design, But the theorem has a loophole.

The reality is that at *any moment in time* you cannot be consistent, available, and partition tolerant.

Riak takes advantage of this fact by allowing you to trade availability for consistency on a per-request basis.

We’ll first look at how Riak clusters its servers and then how to tune reads and writes to interact with the cluster.

The Riak Ring

I will shut down the riak server on 8098 and restart the ring we had last time

for node in dev\*; do $node/bin/riak start; done

Riak divides its server configuration into partitions denoted by a 160-bit number (that’s 2^160).

The Riak team likes to represent this massive integer as a circle, which they call the *ring*.

When a key is hashed to a partition, the ring helps identify which Riak servers store the value.

From Stack Overflow:

Riak uses the word "ring" in two places. The first is to describe the hash space that is used for determining where to store data. The reason Riak calls that space a ring is that the last value in the space (2^160-1) is thought of as being adjacent to the first value in the space (0). Replicas of data are stored in the "next N partitions" of the hash space, following the partition to which the key hashes. Considering the hash space as a ring gives a convenient definition for the "next parition after the final partition."

The other use of the word "ring" is related to, but not exactly the same as the former. I mentioned partitions: each node claims several segments of the hash space, called partitions. Knowledge about which node has claimed which partition is stored in a structure that Riak calls the "ring state", or sometimes just the "ring." Other cluster metadata may also be kept in the ring state, because it's a conveniently shared piece of data throughout the cluster.

In general usage, an application shouldn't need to think about the ring much.

Pasted from <<http://stackoverflow.com/questions/2230088/a-simple-explanation-of-rings-in-riak>>

 One of the first decisions you’ll make when setting up a Riak cluster is how many partitions you’d like.

Let’s consider the case where you have 64 partitions (Riak’s default).

If you divide those sixty-four partitions across three nodes (or, servers), then Riak will give each node twenty-one or twenty-two partitions (64 / 3).

Each partition is called a virtual node, or *vnode*.

Each Riak service counts around the ring on boot, claiming partitions in turn until all vnodes are claimed, as shown in figure 8 "*The Riak ring” of sixty-four vnodes, assigned across three physical node*

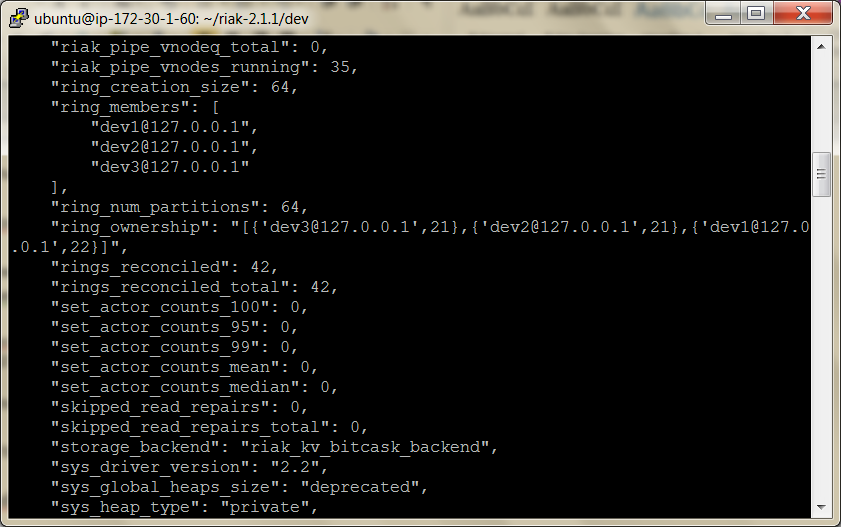
Node A manages vnodes 1, 4, 7, 10...63.

These vnodes are mapped to partitions of the 160-bit ring.

If you view the status of your three development servers

(remember curl -H "Accept: text/plain" [http://localhost:10018/stats](http://localhost:8091/stats) from yesterday), you can see a line like this:

"ring\_ownership": \  
"[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]"



The second number of each object is the number of vnodes each node owns.

They will total sixty-four (21 + 21 + 22).

Each vnode represents a range of hashed keys.

When we insert the room data for key *101*, it may get hashed into the vnode 2 range, so then the key-value object gets stored onto Node B.

The benefit is that if we need to find which server the key lives on, Riak just hashes the key to find the corresponding vnode.

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.

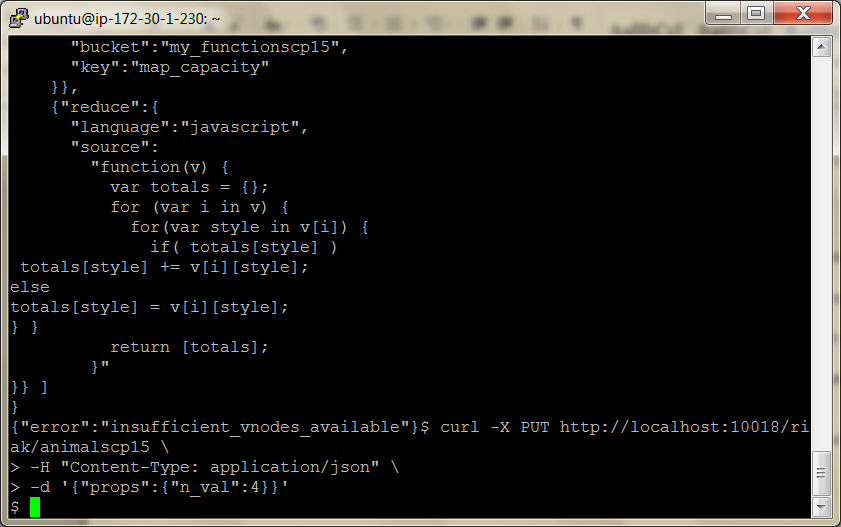
The n\_val bucket property stores the number of nodes to replicate a value to (the N value); it’s 3 by default.

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.

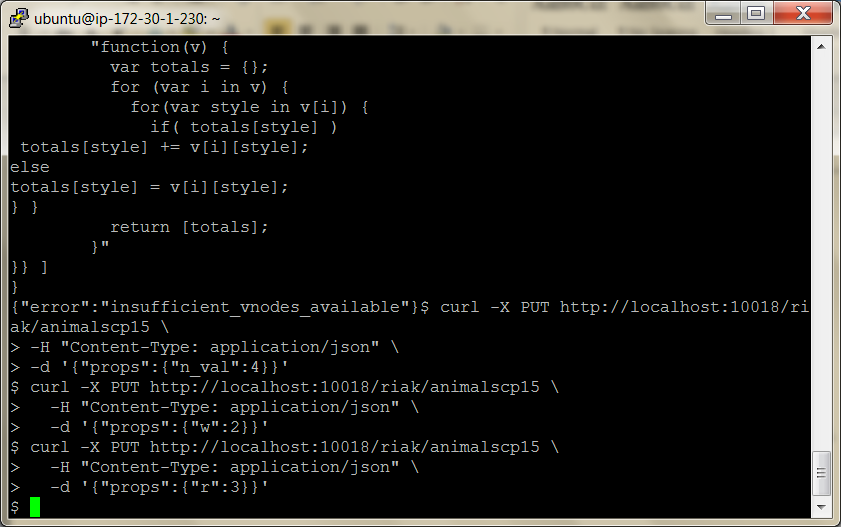
curl -X PUT <http://localhost:10018/riak/pgganimals> \  
 -H "Content-Type: application/json" \  
 -d '{"props":{"w":2}}'

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.

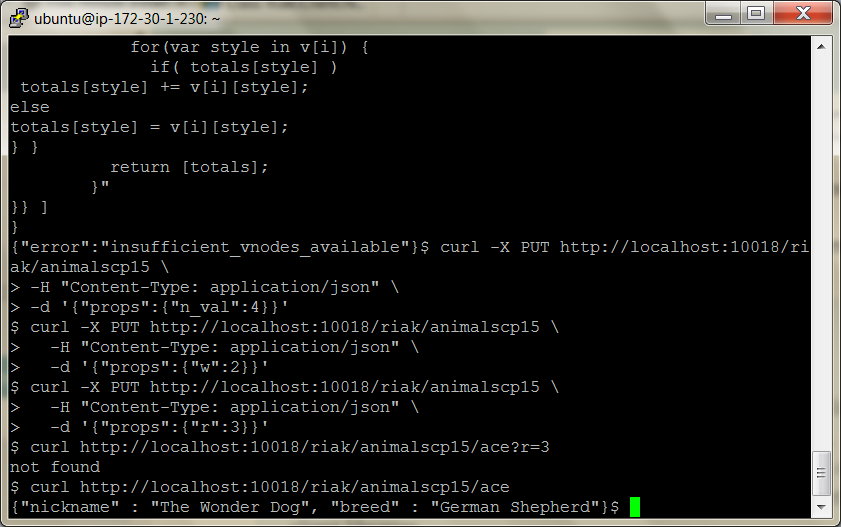
curl -X PUT <http://localhost:10018/riak/pgganimals> \  
 -H "Content-Type: application/json" \  
 -d '{"props":{"r":3}}'



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.

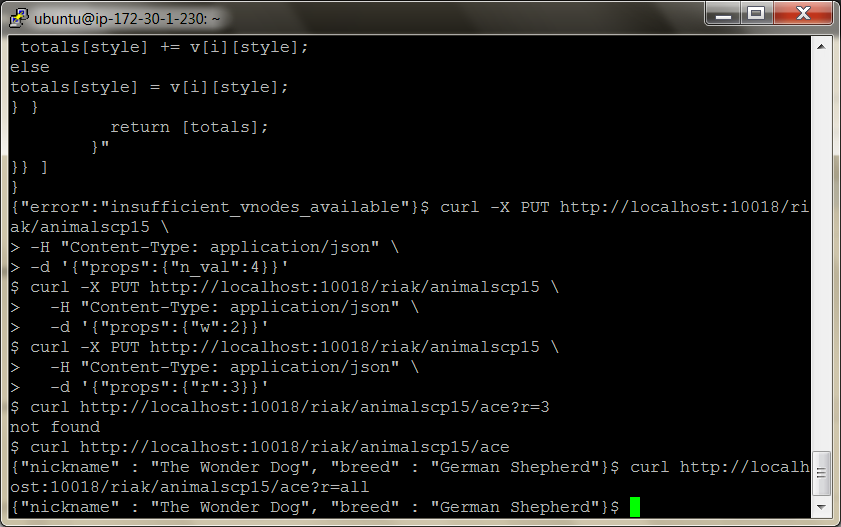
**All** This is the same value as N. Setting W or R to this means all replicated nodes must respond.

**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.

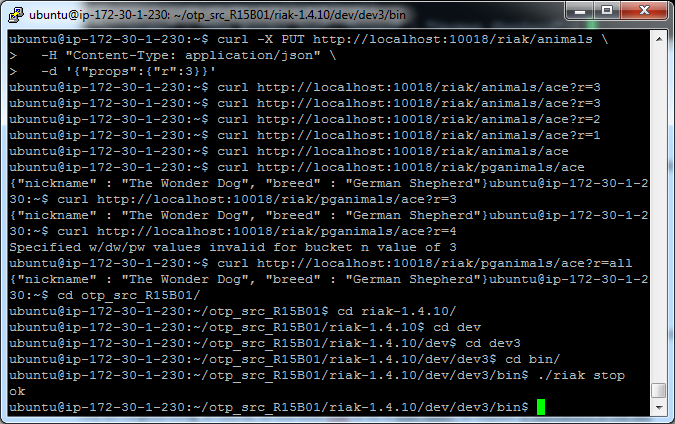
curl <http://localhost:10018/riak/animals/ace?r=all>



The danger with requiring reading from all nodes is that if one goes down, Riak may be unable to fulfill your request.

As an experiment, let’s shut down dev server 3.

**$ dev/dev3/bin/riak stop**



Now if we attempt to read from all nodes, there’s a good chance our request will fail (if it doesn’t, try shutting down dev2 as well, or possibly shut down dev1 and read from port 10028 or 10038; we cannot control what vnodes Riak writes to).

**curl -i** [**http://localhost:10018/riak/animals/ace?r=all**](http://localhost:8091/riak/animals/ace?r=all)

HTTP/1.1 404 Object Not Found

Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic) Date: Thu, 02 Jun 2011 17:18:18 GMT

Content-Type: text/plain

Content-Length: 10

not found

If your request cannot be fulfilled, you’ll get a 404 code (Object Not Found), which makes sense in the scope of the request.

That object cannot be found, because there aren’t enough copies to fulfill the URL request.

This isn’t a good thing, of course, so this kicks Riak to do a *read repair*: to request *N* replications of the key across the servers still available.

If you attempt to access the same URL again, you’ll get the key’s value rather than another 404

But a safer play is to require a quorum (data from most, but not all, vnodes).

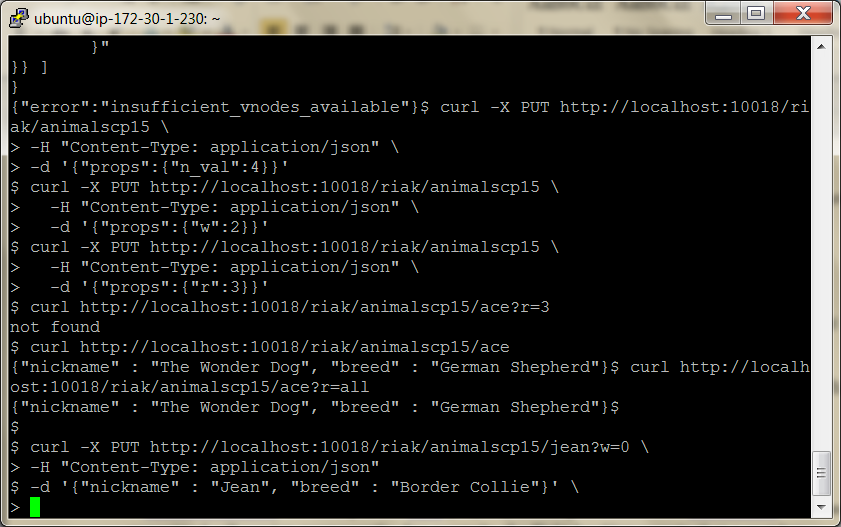
curl <http://localhost:10018/riak/animals/polly?r=quorum>

As long as you write to a quorum, which you can force on a per-write basis, your reads should be consistent.

Another value you can set on-the-fly is W.

If you don’t want to wait for Riak to write to any nodes, you can set W to 0 (zero), which means “I trust you’ll write it, Riak; just return.”

curl -X PUT <http://localhost:10018/riak/animals/jean?w=0> \  
-H "Content-Type: application/json"  
-d '{"nickname" : "Jean", "breed" : "Border Collie"}' \



All this power aside, much of the time you’ll want to stick with the default values unless you have a good reason.

Logs are great for setting W=0, and you can set W=N and R=1 for seldom written data for extra-fast reads.

Writes and Durable Writes

Writes in Riak aren’t necessarily durable, meaning they aren’t immediately written to disk.

(Durability:

In [database systems](https://en.wikipedia.org/wiki/Database_system), **durability** is the [ACID](https://en.wikipedia.org/wiki/ACID) property which guarantees that [transactions](https://en.wikipedia.org/wiki/Database_transaction) that have committed will survive permanently. For example, if a flight booking reports that a seat has successfully been booked, then the seat will remain booked even if the system crashes.

Durability can be achieved by flushing the transaction's log records to [non-volatile storage](https://en.wikipedia.org/wiki/Non-volatile_storage) before acknowledging commitment.

In [distributed transactions](https://en.wikipedia.org/wiki/Distributed_transaction), all participating servers must coordinate before commit can be acknowledged. This is usually done by a [two-phase commit protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol).

Pasted from <<https://en.wikipedia.org/wiki/Durability_(database_systems)>>

)

Although a *node write* may be considered successful, it’s still possible that a failure could occur where a node loses data; even if W=N, servers may fail and lose data.

A write is buffered in memory for a moment before being stored on disk, and that split millisecond is a danger zone.

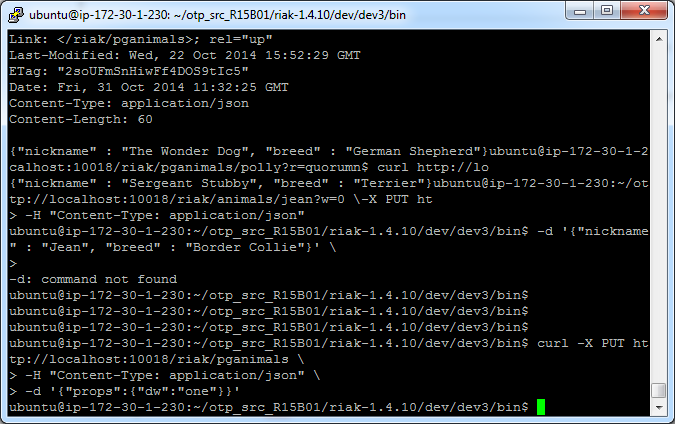
That’s the bad news. The good news is Riak has provided us with a separate setting named DW for *durable write*.

This is slower but further reduces risk, since Riak will not return a success until after the object is written to disk on the given number of nodes.

Just like we did with writes, you can set this property on the bucket.

Here we’re setting dw to be one to be certain at least one node has stored our data.

**$ curl -X PUT** [**http://localhost:10018/riak/animals**](http://localhost:10018/riak/animals) **\  
 -H "Content-Type: application/json" \  
 -d '{"props":{"dw":"one"}}'**



Or, if you like, you can override this on a per-write basis using the dw query parameter in the URL.

A Note on Hinted Handoff

Attempting to write to nodes that aren’t available still succeeds with a “204 No Content.”

This is because Riak will write the value to a nearby node that holds that data until such a time that it can hand it to the unavailable node.

This is a fantastic safety net in the short-term, since if a server goes down, another Riak node will take over.

Of course, if all of server A’s requests get routed to server B, then server B is now dealing with double the load.

There is a danger this will cause B to fail, which might spread to C and D, and so on.

This is known as a *cascading failure*, and it’s rare but possible.

Consider this a fair warning not to tax every Riak server at full capacity, since you never know when one will have to pick up the slack.

HERE – 18/10/18

Resolving Conflicts and Extending Riak

We’ve seen how Riak is a simple key-value database across a cluster of servers.

When dealing with multiple nodes, data conflicts can occur, and sometimes we have to resolve them.

Riak provides a mechanism to sort out which writes happened most recently by way of vector clocks and sibling resolution.

Resolving Conflicts with Vector Clocks

A *vector clock* is a token that distributed systems like Riak use to keep the order of conflicting key-value updates intact.

It’s important to keep track of which updates happen in what order, since several clients may connect to different servers, and while one client updates one server, another client updates another server (you can’t control which server you write to).

You may think “just timestamp the values and let the last value win,” but in a server cluster this works only if all server clocks are perfectly synchronous

Riak makes no such requirement, since keeping clocks synchronized is at best difficult and in many cases an impossible requirement.

Using a centralized clock system would be anathema to the Riak philosophy, since it presents a single point of failure.

Vector clocks help by tagging each key-value event (create, update, or delete) with which client made the change, in which order.

This way, the clients, or application developer, can decide who wins in the case of conflict.

If you are familiar with version control systems like Git or Subversion, this is not dissimilar to resolving version conflicts when two people change the same file.

Vector Clocks in Theory

Let’s say that your dog hotel is doing well so you must start being more selective of the clientele.

To help make the best decision, you’ve gathered a panel of three animal experts to help decide which new dogs are a good fit.

They give each dog a score from 1 (not a good fit) to 4 (a perfect candidate).

All of these panelists—named Bob, Jane, and Rakshith—must reach a unanimous decision.

Each panelist has their own client connecting to a database server, and each client stamps a unique client ID onto each request.

This client ID is used to build the vector clock, as well as keep track of the last updating client in the object header.

Let’s look at a simple pseudocode example and later try the example in Riak.

Bob creates the object first, with a respectable score of 3 for a new puppy named Bruiser.

The vector clock encodes his name and the version 1.

vclock: bob[1]  
value: {score : 3}

Jane pulls this record and gives Bruiser a score of 2.

The vclock created for her update succeeded Bob’s, so her version 1 is added to the end of the vector.

vclock: bob[1], jane[1]  
value: {score : 2}

Simultaneously, Rakshith pulled the version that Bob created but not Jane’s.

He loved Bruiser and set a score of 4.

Just like Jane’s, his client name is appended to the end of the vector clock as version 1.

vclock: bob[1], rakshith[1]  
value: {score : 4}

Later that day, Jane (as the panel chair) rechecks the scores.

Since Rakshith’s update vector did not occur after Jane’s but rather alongside hers, the updates are in conflict and need to be resolved.

She receives both values, and it’s up to her to resolve them.

vclock: bob[1], jane[1]  
value: {score : 2}  
---  
vclock: bob[1], rakshith[1]  
value: {score : 4}

She chooses a middle value so updates the score to 3.

vclock: bob[1], rakshith[1], jane[2]  
value: {score : 3}

Having been resolved, anyone who pulls a request after this point will get this most recent value.

Vector Clocks in Practice

Let’s run through the previous example scenario using Riak.

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

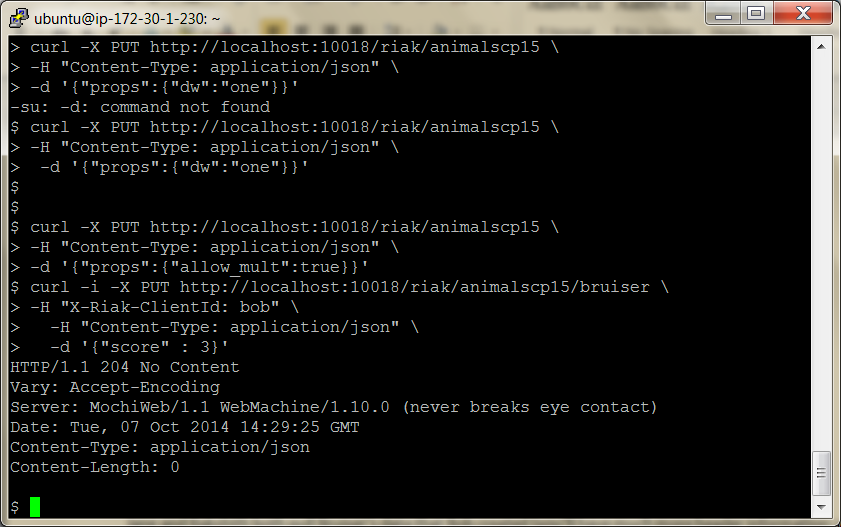
Let’s keep multiple versions by setting the allow\_mult property on the animals bucket.

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

$ curl -X PUT <http://localhost:10018/riak/animals> \  
 -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*.

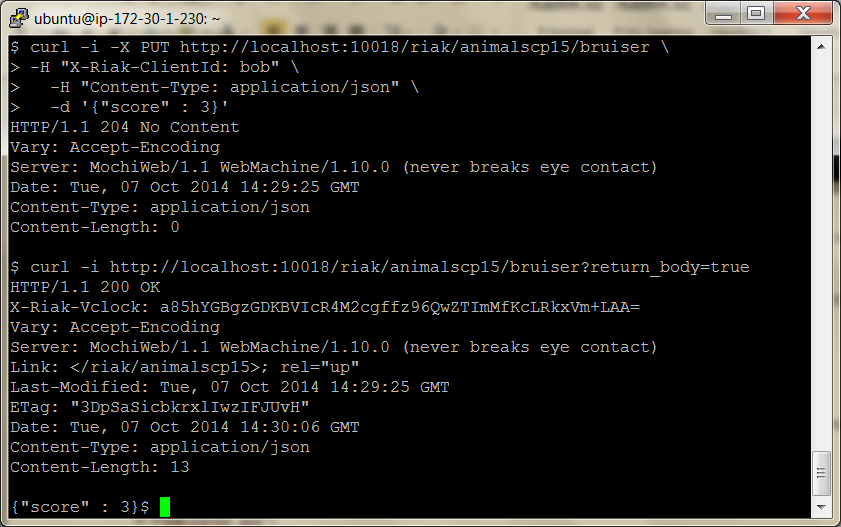
$ curl -i -X PUT <http://localhost:10018/riak/animals/bruiser> \  
 -H "X-Riak-ClientId: bob" \  
 -H "Content-Type: application/json" \  
 -d '{"score" : 3}'



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/animals/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/animals/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/animals/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/animals/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/animals/bruiser?return_body=true> \  
 -H "accept: multipart/mixed"

HTTP/1.1 300 Multiple Choices  
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA  
Content-Type: multipart/mixed; boundary=1QwWn1ntX3gZmYQVBG6mAZRVXlu  
Content-Length: 409

--1QwWn1ntX3gZmYQVBG6mAZRVXlu  
Content-Type: application/json  
Etag: 637aZSiky628lx1YrstzH5

{"score" : 4}  
--1QwWn1ntX3gZmYQVBG6mAZRVXlu  
Content-Type: application/json  
Etag: 7F85FBAIW8eiD9ubsBAeVk

{"score" : 2}  
--1QwWn1ntX3gZmYQVBG6mAZRVXlu--

Notice that the “siblings” shown earlier are HTTP etags (which Riak called vtags) to specific values.

As a side note, you can use the Etag parameter in the URL to retrieve only that version:

curl <http://localhost:10018/riak/animals/bruis>er?vtag=7F85FBAIW8eiD9ubsBAeVk

will return {"score" : 2}.

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/animals/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/animals/bruiser?return_body=true>

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

{"score" : 3}

Any future requests will receive score 3.

Time Keeps on Growing

You may have noticed that the vector clock keeps growing as more clients update values.

This is a fundamental problem with vector clocks, which the Riak developers understood.

They extended vector clocks to be “pruned” over time, thus keeping their size small.

The rate at which Riak prunes old vector clock values are bucket properties, which can be viewed (along with all other properties) by reading the bucket.

$ curl <http://localhost:10018/riak/animals>

You’ll see some of the following properties, which dictate how Riak will prune the clock before it gets too large.

"small\_vclock":10,"big\_vclock":50,"young\_vclock":20,"old\_vclock":86400

*small\_vclock* and *big\_vclock* determine the minimum and maximum length of the vector, while *young\_vclock* and *old\_vclock* describe the minimum and maximum age of a vclock before pruning happens

 Wrap-Up

Riak is a distributed, data- replicating, enhanced key-value store without a single point of failure.

If your experience with databases until now has been only relational, Riak may seem like an alien beast.

There are no transactions, no SQL, no schema.

There are keys, but linking between buckets is not at all like a table join, and mapreduce can be a daunting methodology

The trade-offs, however, are worth it for a certain class of problems.

Riak’s ability to scale out with more servers (rather than scale up with larger single servers) and its ease of use are excellent attempts at solving the unique scalability problems of the Web.

And rather than reinventing the wheel, Riak piggybacks on the HTTP structure, allowing maximum flexibility for any framework or web-enabled system.

Riak’s Strengths

If you want to design a large-scale ordering system a la Amazon, or in any situation where high availability is your paramount concern, you should consider Riak.

Hands down, one of Riak’s strengths lies in its focus on removing single points of failure in an attempt to support maximum uptime and grow (or shrink) to meet changing demands.

If you do not have complex data, Riak keeps things simple but still allows for some pretty sophisticated data diving should you need it.

There is currently support for about a dozen languages (which you can find on the Riak website) but is extendable to its core if you like to write in Erlang.

And if you require more speed than HTTP can handle, you can also try your hand at communicating via Protobuf, which is a more efficient binary encoding and transport protocol.

Riak’s Weaknesses

If you require simple queryability, complex data structures, or a rigid schema or if you have no need to scale horizontally with your servers, Riak is probably not your best choice.

One of our major gripes about Riak is it still lags in terms of an easy and robust ad hoc querying framework, although it is certainly on the right track.

Mapreduce provides fantastic and powerful functionality, but we’d like to see more built-in URL-based or other PUT query actions.

The addition of indexing was a major step in the right direction and a concept we’d love to see expanded upon.

Finally, if you don’t want to write Erlang, you may see a few limitations using JavaScript, such as the unavailability of post-commit or slower mapreduce execution.

However, the Riak team is working on these relatively minor hiccups.

Riak on CAP

Riak provides a clever way of circumventing the constraints that CAP places on all distributed databases.

How it dances around the problem is astounding, compared to a system like PostgreSQL that can (largely) only support strong write consistency.

Riak leverages the Amazon Dynamo paper’s insight that CAP can be changed on a per-bucket, or per-request, basis.

It’s a big step forward for robust and flexible open source database systems.

As you read about other databases, keep Riak in mind, and you’ll continue to be impressed by its flexibility.

Parting Thoughts

If you need to store a huge catalog of data, you could do worse than Riak.

Though relational databases have been researched and tweaked for more than forty years, not every problem needs ACID compliance or the ability to enforce a schema.

If you want to embed a database into a device or handle financial transactions, you should avoid Riak.

If you want to scale out or serve up loads of data on the Web, take a look.

 Notes:

To add/remove users

sudo useradd -d /home/blove -m blove -G admin

sudo userdel user

sudo rm -r /home//

Launch server in Ec2, log in using putty, use nosql16.ppk for public keys