

1. Riak	2
1.1 The Riak Fast Track	2
1.1.1 What is Riak?	3
1.1.2 Building a Development Environment	5
1.1.3 Basic Riak API Operations	8
1.1.4 Loading Data and Running MapReduce Queries	11
1.1.5 Tunable CAP Controls in Riak	16
1.1.6 Thank You	17
1.2 Installation and Setup	17
1.2.1 Installing Erlang	18
1.2.2 Installing on Debian and Ubuntu	20
1.2.3 Installing on Mac OS X	20
1.2.4 Installing on RHEL and CentOS	21
1.2.5 Installing Riak from Source	22
1.2.6 Installing with Chef	23
1.2.7 Basic Cluster Setup	23
1.2.8 Setting Up Innostore	24
1.3 An Introduction to Riak	27
1.4 How Things Work	31
1.4.1 Replication	32
1.4.2 REST API	33
1.4.3 Links	44
1.4.4 MapReduce	45
1.4.5 Configuration Files	52
1.4.6 PBC API	55
1.4.7 Erlang Client PBC	67
1.4.8 Pre- and Post-Commit Hooks	71
1.5 Client Libraries	73
1.5.1 Community-Developed Libraries and Projects	74
1.6 Riak Comparisons	75
1.6.1 Riak Compared to Cassandra	75
1.6.2 Riak Compared to MongoDB	77
1.6.3 Riak Compared to Neo4j	78
1.7 FAQ	80
1.8 Recommended Resources	81
1.9 Riak Glossary	82
1.10 Sample Data	84
1.11 Best Practices	84
1.11.1 Hosting and Server Configuration	85
1.11.2 Innostore Configuration and Tuning	85
1.11.3 Network Security and Firewall Configurations	87
1.12 Benchmarking with Basho Bench	87
1.13 Riak Recaps	94
1.14 Contact Basho	95

Riak

Riak: An Open Source Internet-Scale Data Store

Riak is a Dynamo-inspired key/value store that scales predictably and easily. Riak also simplifies development by giving developers the ability to quickly prototype, test, and deploy their applications.

A truly fault-tolerant system, Riak has no single point of failure. No machines are special or central in Riak, so developers and operations professionals can decide exactly how fault-tolerant they want and need their applications to be.

Get Riak

- [Download a Riak package](#)
- [Build Riak on your preferred Operating System](#)
- [Get the source](#)

Connect your Code

Looking for a specific Riak Driver? Basho currently supports libraries for [Erlang](#), [Javascript](#), [Java](#), [PHP](#), [Python](#) and [Ruby](#).

There's also a bunch of [community contributed code and projects](#) and it's easy to connect to Riak via the [REST API](#).

New to Riak? Get on the Fast Track!

The [Fast Track](#) is a 30-45 minute interactive tutorial that introduces the basics of using and understanding Riak, from building a three node cluster up through MapReduce.

Learn More

- Get a [high-level overview](#) of Riak's architecture.
- Learn [in-depth](#) how Riak functions.
- See [screencasts](#), [blog posts](#), [sample applications](#) and more.
- Browse the [Frequently Asked Questions](#).



The Riak Fast Track

- [What is the Riak Fast Track?](#)
- [Who should use the Riak Fast Track?](#)
- [What does the Fast Track Cover?](#)
- [How does it work?](#)
- [How long will the Fast Track take?](#)

What is the Riak Fast Track?

We wanted a straightforward, concise, interactive module to acquaint people with Riak and how it works. We figured one of the best ways to do this was to make it easy to learn by doing. So, the Fast Track is a five part module that enables people to do just that: get a grasp on some of Riak's core concepts quickly.

Who should use the Riak Fast Track?

The Fast Track is aimed at people who have little or no experience with Riak. That said, even if you're already well-versed in the ways of Riak or have a decent understanding of it, we would encourage you to go through this and try it; you may find some use in this, too. Also, the more eyeballs we throw at it, the more we can refine and simplify it for the generations of new users to come.

What does the Fast Track Cover?

The Fast Track will take you through the following five sections:

1. A High Level Overview of Riak and its Architecture
2. Setting up a Development Cluster on your Machine
3. Standard API Operations
4. Importing Data and Running Simple MapReduce Queries
5. Tunable CAP Controls in Riak

(Note: This by no means covers everything that Riak can do, but it should help you get a decent handle on some core functionality)

How does it work?

The Fast Track is broken up into five, one page sections. Start with the first section, and when you get to the end, follow the [What's Next?](#) link located at the bottom of each page. This will take you to the next section in the tutorial. Simply repeat this process until you reach the end.

You will also notice that at the bottom of each page there is a section entitled "Additional Reading." These are pointers to resources that might be worthwhile if you want to learn more about what's being discussed in each section.

How long will the Fast Track take?

This module was intended to be consumed in one sitting (though nothing will be lost if you finish, say, 30% and pick it back up down the road). From start to finish, this will probably take you around 45 minutes. (Trust us, it's worth it.)

So, if you have an hour and want to get your hands dirty with Riak, go through the Riak Fast Track and try your hand at using it. Upon completion, you should have a decent grasp of the mechanisms at work and the features present in a Riak development environment. (If you don't, please email us or post in the comments. We want to know what you think.)

What's Next? ----> [What is Riak?](#)

What is Riak?

Riak is:

- A Database
- A Data Store
- A key/value store
- Used by Fortune 100 Companies
- Used by startups
- A "NoSQL" database
- Schemaless and data-type agnostic
- Written (primarily) in Erlang
- As distributed as you want and need it to be
- Scalable
- Pronounced "REE-ack"
- Not the best fit for every project and application
- And much, much more...

Here are some technical specifics that you should know before diving into the Fast Track:

Data

Buckets, Keys, Values

Riak organizes data into Buckets, Keys, and Values. Values (or objects) are identifiable by a unique key, and each key/value pair is stored in a bucket.

Buckets are essentially a flat namespace in Riak and have little significance beyond their ability to allow the same key name to exist in multiple buckets and to provide some per-bucket configurability for things like replication factor and pre/post-commit hooks.

The Riak API

We initially implemented both a native Erlang interface, and a HTTP (often referred to as "RESTful") API that allowed users to manipulate data using standard HTTP methods: GET, PUT, UPDATE and DELETE.

With the 0.10 release, we added support for accessing Riak using a Protocol Buffers Client interface.

Versioning

Each update to a Riak object is tracked by a vector clock. Vector clocks determine causal ordering and detect conflicts in a distributed system. Each time a key/value pair is created or updated in Riak, a vector clock is generated to keep track of each version and ensure that the proper value can be determined in the event that there are conflicting updates.

Riak has two ways of resolving update conflicts on Riak objects. Riak can allow the last update to automatically "win" or Riak can return both versions of the object to the client. This gives the client the opportunity to resolve the conflict on its own.

Languages

At the moment, the core Basho Development Team currently supports libraries for:

- Erlang
- JavaScript
- Java
- PHP
- Python
- Ruby

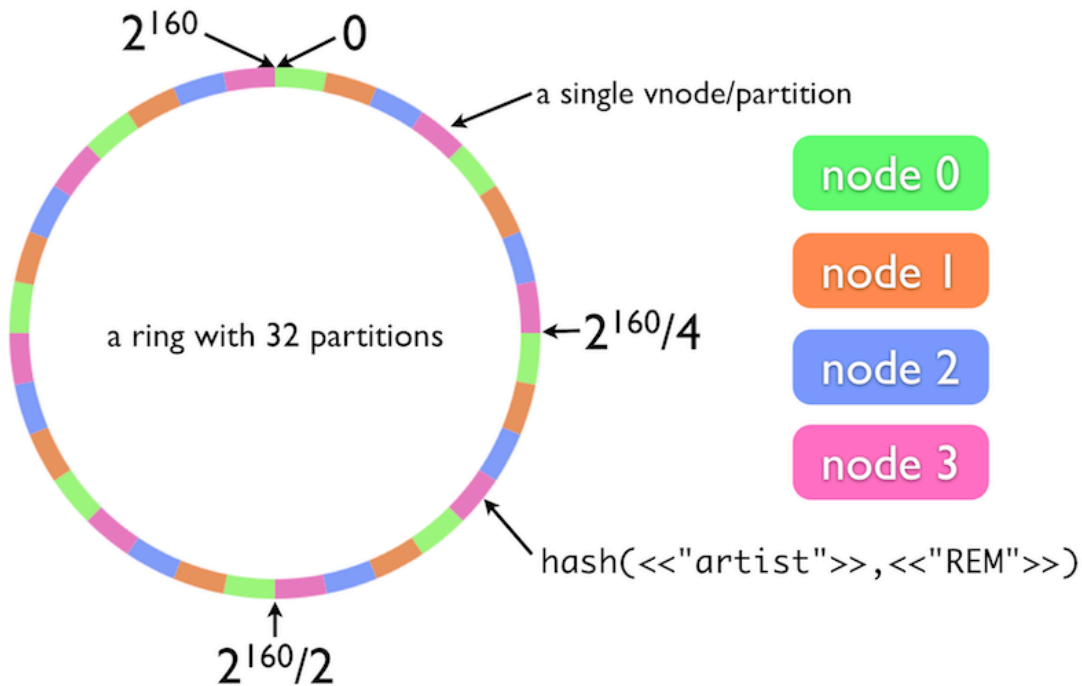
In addition, there are community contributed projects for .NET, JavaScript, Python (and Twisted), Griffon, Perl, and Scala. (And this list is growing rapidly.)

The Riak Cluster

Central to any Riak cluster is a 160-bit integer space (often referred to as "the ring") which is divided into equally-sized partitions.

Physical servers, referred to in the cluster as "nodes," run a certain number of virtual nodes, or "vnodes". Each vnode will claim a partition on the ring. The number of active vnodes is determined by the number of partitions into which the ring has been split. This is a static number that is chosen at cluster initialization.

As a rule, each node in the cluster is responsible for $1/(\text{total number of physical nodes})$ of the ring. You can determine the number of vnodes on each node by calculating $(\text{number of partitions})/(\text{number of nodes})$. More simply put, a ring with 32 partitions, composed of four physical nodes, will have approximately eight vnodes per node. This setup is represented in the diagram below.



All nodes in a Riak cluster are equal. Each node is fully capable of serving any client request. This is possible due to the way Riak uses consistent hashing to distribute data around the cluster.

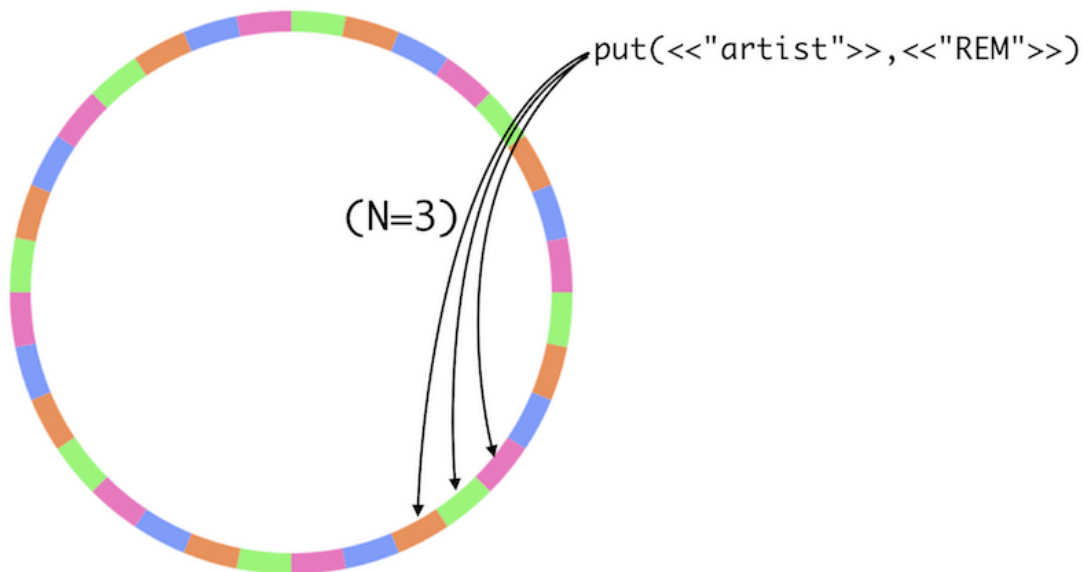
A Riak cluster grows and shrinks dynamically, meaning Riak will automatically re-balance data as nodes join and leave the cluster.

Data Replication

Replication is fundamental and automatic in Riak, providing security that your data will still be there if a node in your Riak cluster goes down. All data stored in Riak will be replicated to a number of nodes in the cluster according to the `n_val` property set on the bucket.

For example, here is a depiction of what happens when `n_val = 3` (This is the default setting). When you store a datum in a bucket with an `N`

value of three, the datum will replicated to three separate partitions on the Riak Ring.



Querying and Query Languages

At the moment, Riak relies on MapReduce to perform queries that exceed the limitations that come with basic key/value storage model.

Currently users can write their MapReduce queries in either Erlang or Javascript.

What's Next? -> You know some of the basics. [Now it's time to build a three node Riak cluster.](#)



Additional Reading

- [A High Level Introduction to Riak](#)
- [The Riak Glossary](#)
- [More on Vector Clocks](#)
- [Protocol Buffers Client Interface](#)
- [Client Libraries](#)
- [Replication](#)
- [Recommended General Resources](#)

Building a Development Environment

One of the first advantages of Riak that should be highlighted is that it allows for quick prototyping and development. Unlike some databases, with Riak, you can spin up (and spin down) a large number of Riak nodes to use on your local environment when you are planning, testing, and building your application.

In this section, we will take you through the process to build a development environment. It's simple, and shouldn't take you more than a few minutes. Once you are done, you will have a three node Riak cluster running on your local machine.

To make it as easy as possible, we've recorded a screencast. If you don't like videos and prefer to read, the instructions below mirror those that are in the screencast. Either way you choose, you'll have a three node Riak cluster in no time!



A few things to note before we start

- You do not need to download and build Erlang from source to install Riak. We have pre-packaged binaries available for most major platforms that embed the Erlang runtime. However, for development/source builds, you'll need to have Erlang R13B04 or later, and this tutorial is based on a source build.
- The setup outlined below that you are about to build sets up nodes with HTTP interfaces listening on ports 8091-3. The default port for nodes to listen on is *8098* and users will need to take note of this when trying to use any of the default other-language client settings.

Downloading Riak and Building a Three Node Cluster

Download and install the latest Erlang

Running Riak requires Erlang R13B04 or later. We have platform specific instructions written up for downloading Erlang located [here](#). If you don't already have Version R13B04 installed, go do so and hurry back.

Download Mercurial

Mercurial is the source control system that Basho uses for Riak development.

To download this, you can use either `easy_install` or `pip`, or head to <http://mercurial.selenic.com/> and follow their simple instructions.



Basho also mirrors its various repositories at GitHub, so if you already have Git installed, you can skip this step and follow the Git instructions in the next section.

Clone a copy of the latest Riak from source

Now that you have Erlang and Mercurial squared away, it's time to clone the latest Riak. From a terminal, run the following command:

```
$ hg clone http://bitbucket.org/basho/riak
```

If you prefer to clone from GitHub:

```
$ git clone git://github.com/basho/riak.git
```

You should now have the latest version of Riak.

Build Riak

So now you have a copy of Riak. Time to build it. Do this using by accessing the "riak" directory

```
$ cd riak
```

and running "make all"

```
$ make all
```

As you can see, "make all" is grabbing all the Riak dependencies for you so that you don't have to chase them down. This should take a few moments.

Use Rebar to start up three nodes

Now that Riak is built, we are going to use Rebar, a packaging and build system for Erlang applications, to get three self-contained Riak nodes running on your machine. Tomorrow, when you put Riak into production, Rebar will enable you to ship a pre-built Riak package to your deployment machines. But for now, we will just stick to the three nodes. To start these up, run "make devrel"

```
$ make devrel
```

You have just generated a "dev" directory. Let's go into that directory to check out its contents:

```
$ cd dev; ls
```

That should give you the following:

```
dev1      dev2      dev3      overlay  reltool.config riak
```

Each item starting with "dev" is a complete package containing a Riak node. We now need to start each node. Let's start with "dev1"

```
$ dev1/bin/riak start
```

Then do the same for "dev2" and "dev3"

```
$ dev2/bin/riak start
```

```
$ dev3/bin/riak start
```

Test to see the running Riak nodes

After you have the nodes up and running, it's time to test them and make sure they are available. You can do this by taking a quick look at your process list. To do this, run:

```
$ ps aux | grep beam
```

This should give you details on three running Riak nodes.

Join the nodes to make a cluster

The next step is join these three nodes together to form a cluster. You can do this using the Riak Admin tool. Specifically, what we want to do is join "dev2" and "dev3" to "dev1":

```
$ dev2/bin/riak-admin join dev1@127.0.0.1
```

followed by

```
$ dev3/bin/riak-admin join dev1@127.0.0.1
```

Test the cluster and add some data to verify the cluster is working

Great. We now have a running three node Riak cluster. Let's make sure it's working correctly. For this we can hit Riak's HTTP interface using *curl*. Try this:

```
$ curl -H "Accept: text/plain" http://127.0.0.1:8091/stats
```

Once this runs, look for a field in the output named "ring_ownership." This should show you that all three of your nodes are part of your Riak cluster. You might also notice that each node has claimed a set of partitions. The default partition setting for our three node development cluster is 64. This means that two of the three nodes will have claimed 21 partitions, and the third node will handle the last 22. (The number of partitions in your cluster is set with "ring_creation_size" parameter in Riak's app.config file.)

If you want, you can add a file to your Riak cluster and test it's working properly. Let's say, for instance, we wanted to add an image and make sure it was accessible. First, copy an image to your directory if you don't already have one:

```
$ cp ~/image/location/image_name.jpg .
```

We can then PUT that image into Riak using a curl command:

```
$ curl -X PUT HTTP://127.0.0.1:8091/riak/images/1.jpg -H "Content-type: image/jpeg" --data-binary @image_name.jpg
```

You can then verify that image was in fact stored. To do this, simply copy the URL where we PUT the image and paste it into a browser. Your image should appear.

You should now have a running, three node Riak cluster. Congratulations! That didn't take so long, did it?

What's Next? --> You now have a three node Riak Cluster up and running. Time to learn about the basic HTTP API operations.

Additional Reading

- Platform Specific Riak Builds
- [Rebar](#)

Basic Riak API Operations

Riak currently offers three ways to interact with the database:

1. HTTP Interface (often referred to as RESTful)
2. Native Erlang Interface
3. Protocol Buffers (PB) Interface

For this tutorial, we are going to work with the HTTP Interface and use `curl` commands through the terminal to test Riak out and help you get acquainted with it quickly. (You can find links to details about the Erlang and PB interfaces at the bottom of this page.)

HTTP API Operations

- Object/Key Operations
 - Read an Object
 - Store an object with existing or user-defined key
 - Store a new object and assign random key
 - Delete an object
- Bucket Properties and Operations
 - Setting a bucket's properties
 - GET Buckets



Required Knowledge

- **Client ID** - All requests should include the `X-Riak-ClientId` header, which can be any string that uniquely identifies the client, for purposes of tracing object modifications in the [vector clock](#).
- **URL Escaping** - Buckets, keys, and link specifications may not contain unescaped slashes. Use a URL-escaping library or replace slashes with `%2F`.

Object/Key Operations

Most of the interactions you'll have with Riak will be setting or retrieving the value of a key. This section describes how to do that.

Read an Object

Here is the basic command formation for retrieving a specific key from a bucket.

```
GET /riak/bucket/key
```

The body of the response will be the contents of the object (if it exists). Simple, right?

Riak understands many HTTP-defined headers, like `Accept` for content-type negotiation (relevant when dealing with multi-version "sibling" objects, see [REST API#Siblings examples](#)), and `If-None-Match/ETag` and `If-Modified-Since/Last-Modified` for conditional requests.

Riak also accepts many query parameters, including `r` for setting the R-value for this get request (R Values will be explained more in the final section of the Fast Track Tutorial). Omitting the `r` query parameter is equivalent to specifying `r=2`.

Normal response codes: 200 OK, 300 Multiple Choices, 304 Not Modified

Typical error codes: 404 Not Found

So, with that in mind, try this command. This will request (GET) the key "doc2" from the bucket "test."

```
$ curl -v http://127.0.0.1:8091/riak/test/doc2
```

This should return a "404 Not Found" as the key "doc2" does not exist, but that's not a bad thing. This is the response we wanted to receive.

Store an object with existing or user-defined key

Your application will often have its own method of generating the keys for its data. If so, storing that data is easy. The basic request looks like this.

```
PUT /riak/bucket/key
```




POST is also a valid verb, for compatibility's sake.

Some request headers are required for PUTs:

- `Content-Type` must be set for the stored object. Set what you expect to receive back when next requesting it.
- `X-Riak-Vclock` if the object already exists, the vector clock attached to the object when read; if the object is new, this header may be omitted

Other request headers are optional for PUTs:

- `X-Riak-Meta-*` any additional metadata headers that should be stored with the object.
- `Link` - user- and system-defined links to other resources. [Read more about Links](#)

Similar to how GET requests support the "r" query parameter, PUT requests also support these parameters:

- `r` how many replicas need to agree when retrieving an existing object before the write (integer value, default is 2)
- `w` how many replicas to write to before returning a successful response (integer value, default is 2).
- `dw` how many replicas to commit to durable storage before returning a successful response (integer value, default is 0)
- `returnbody` whether to return the contents of the stored object (boolean string, default is "false")

Normal status codes: 200 OK, 204 No Content, 300 Multiple Choices

If `returnbody=true`, any of the response headers expected from [Read object](#) may be present. Like [Read object](#), 300 Multiple Choices may be returned if siblings existed or were created as part of the operation, and the response can be dealt with similarly.

Let's give it a shot. Try running this in a terminal.

```
$ curl -v -X PUT -d '{"bar":"baz"}' -H "Content-Type: application/json" -H "X-Riak-Vclock: a85hYGBgzGDKBVIszMk55zKYEhznzWBlKIniO8mUBAA==" http://127.0.0.1:8091/riak/test/doc?returnbody=true
```

Store a new object and assign random key

If your application would rather leave key-generation up to Riak, issue a POST request to the bucket URL instead of a PUT to a bucket/key pair:

```
POST /riak/bucket
```

If you don't pass Riak a "key" name after the bucket, it will know to create one for you.

Supported headers are the same as for bucket/key PUT requests, though `X-Riak-Vclock` will never be relevant for these POST requests. Supported query parameters are also the same as for bucket/key PUT requests.

Normal status codes: 201 Created

This command will store an object, in the bucket "test" and assign it a key:

```
$ curl -v -d 'this is a test' -H "Content-Type: text/plain" http://127.0.0.1:8091/riak/test
```

In the output, the "Location" header will give you the key for that object. To view the newly created object, go to "<http://127.0.0.1:8091/<Location>>" in your browser.

If you've done it correctly, you should see the value (which is "this is a test").

Delete an object

Lastly, you'll need to know how to delete keys.

The command, as you can probably guess, follows a predictable pattern and looks like this:

```
DELETE /riak/bucket/key
```

The normal response codes for a DELETE operations are 204 No Content and 404 Not Found

404 responses are "normal" in the sense that DELETE operations are idempotent and not finding the resource has the same effect as deleting it.

Try this:

```
$ curl -v -X DELETE http://127.0.0.1:8091/riak/test/test2
```

Bucket Properties and Operations

Buckets are essentially a flat namespace in Riak and have little significance beyond their ability to allow the same key name to exist in multiple buckets and to provide some per-bucket configurability.

Setting a bucket's properties

There is no need to "create" buckets in Riak. They pop into existence when keys are added to them, and disappear when all keys have been removed from them.

However, in addition to providing a namespace for keys, the properties of a bucket also define some of the behaviors that Riak will implement for the values stored in the bucket. You may set the properties of a bucket before keys are stored in that bucket; Riak will remember them (indeed for some like `n_val`, you are encouraged to do so).

To set these properties, issue a PUT to the bucket's URL:

```
PUT /riak/bucket
```

The body of the request should be a JSON object with a single entry "props." Unmodified bucket properties may be omitted.

Important headers:

- `Content-Type: application/json`

The most important properties to consider for your bucket are:

- `n_val`: the number of replicas for objects in this bucket (defaults to "3"); `n_val` should be an integer greater than 0 and less than the number of partitions in the ring. (*Changing `n_val` after it is initially set is not advisable as it may result in failed reads because the new value may not be replicated to all the appropriate partitions.*)
- `allow_mult = true` or `false` (Defaults to `false`); Riak maintains any sibling objects caused by things like concurrent writers of network partitions. With `allow_mult` set to `false`, clients will only get the most-recent-by-timestamp object.

Let's go head and alter the properties of a Bucket. The following PUT will create a new bucket called "test" with a modified `n_val` of 5.

```
$ curl -v -X PUT -H "Content-Type: application/json" -d '{"props":{"n_val":5}}' http://127.0.0.1:8091/riak/test
```

GET Buckets

Here is how you use the HTTP API to retrieve (or "GET") the bucket properties and/or keys:

```
GET /riak/bucket_name
```

Again, quite simple. (Are you starting to see a pattern?)

The optional query parameters are:

- `props=[true|false]` - whether to return the bucket properties (defaults to "true")
- `keys=[true|false|stream]` - whether to return the keys stored in the bucket (defaults to "true"); see [REST API#Read bucket information](#) for details about dealing with a `keys=stream` response

With that in mind, go ahead and run this command. This will GET the bucket information that we just set with the sample command above:

```
$ curl -v http://127.0.0.1:8091/riak/test
```

You can also view this Bucket information through any browser by going to "http://127.0.0.1:8091/riak/test"

So, that's how the REST API works. There are more nuances to what you can do with it, and an in depth reading of the REST API page (linked below) is highly recommended. This will give you details on the headers, parameters, and status that you should keep in mind when using the HTTP Interface. But, for the purpose of the Fast Track, this whirlwind tour will be sufficient.

What's Next? -> Now that we have worked a bit with the HTTP interface to Riak, let's load some sample data and run some more advanced queries using MapReduce.



Additional Reading for this Section

- [The REST API In Depth](#)
- [Protocol Buffers API](#)
- [Replication in Depth](#)
- [Why Vector Clocks are Easy](#) (from the Basho Blog)
- [Why Vector Clocks are Hard](#) (from the Basho Blog)

Loading Data and Running MapReduce Queries

This section will walk you through loading some sample data (that we've borrowed from Google) into Riak and then using JSON over HTTP interface with Curl to perform some MapReduce queries on that data.

Sample Data

This Erlang script will load historical stock-price data for Google (ticker symbol "GOOG") into your existing Riak cluster so we can use it. Paste the code below into a file called `load_data` inside the `dev` directory (or download it below).

```

                                load_data
-----
#!/usr/bin/env escript
%% -*- erlang -*-
main([Filename]) ->
    {ok, Data} = file:read_file(Filename),
    Lines = tl(re:split(Data, "\r?\n", [{return, binary},trim])),
    lists:foreach(fun(L) -> LS = re:split(L, ","), format_and_insert(LS) end, Lines).

format_and_insert(Line) ->
    JSON = io_lib:format(
    {"Date\":"~s\","Open\":"~s\","High\":"~s\","Low\":"~s\","Close\":"~s\","Volume\":"~s\","Adj.
Close\":"~s"}, Line),
    Command = io_lib:format("curl -X PUT http://127.0.0.1:8091/riak/goog/~s -d '~s' -H
'content-type: application/json'", [hd(Line),JSON]),
    io:format("Inserting: ~s~n", [hd(Line)]),
    os:cmd(Command).

```

Make the script executable:

```
$ chmod +x load_data
```

Download the CSV file of stock data linked below and place it in the `dev` directory where we've been working.

Name	Size	Creator	Creation Date	Comment
goog.csv	76 kB	Sean Cribbs	May 06, 2010 10:55	Google historical stock data
load_stocks.rb	0.3 kB	Sean Cribbs	May 06, 2010 10:33	Alternative script in Ruby to load the data
load_data	0.6 kB	Sean Cribbs	May 17, 2010 16:15	Erlang script to load data (as shown in snippet)

Now load the data into Riak.

```
$ ./load_data goog.csv
```

So now we have some data in our Riak cluster. Let's put that aside for a minute and learn a bit about MapReduce, and how Riak uses it.

MapReduce

MapReduce is a programming paradigm, popularized by Google. In Riak, MapReduce is the primary method for non-primary-key-based querying.

Riak enables you to run MapReduce jobs through both the Erlang API and the REST API. For this tutorial we are going to use the REST API.

Why do we use MapReduce for Querying Riak?

Key-value stores like Riak generally have very little functionality beyond just storing and fetching objects. MapReduce adds the capability to perform more powerful queries over the data stored in Riak. It also fits nicely with the functional programming orientation of Riak's core code and the distributed nature of the data storage.

The main goal of MapReduce is to spread the processing of a query across many systems to take advantage of parallel processing power. This is generally done by dividing the query into several steps, dividing the dataset into several chunks, and then running those step/chunk pairs on separate physical hosts. Riak's MapReduce has an additional goal: increasing data-locality. When processing a large dataset, it's often much more efficient to take the computation to the data than it is to bring the data to the computation.

"Map" and "Reduce" are both phases in the query process. Map functions take one piece of data as input, and produce zero or more results as output. If you're familiar with "mapping over a list" in functional programming style, you're already familiar with "map" steps in a map/reduce query.

REST Query Syntax

Before we run some MapReduce queries of our own on the sample data, we should review a bit about how to write the queries and how they are executed.

MapReduce queries are issued over HTTP via a POST to the `/mapred` resource. The body should be `application/json` of the form {
"inputs": [...inputs...], "query": [...query...]}.

Map/Reduce queries have a default timeout of 60000 milliseconds (60 seconds). The default timeout can be overridden by supplying a different value, in milliseconds, in the JSON document {"inputs": [...inputs...], "query": [...query...], "timeout": 90000}

Inputs

The list of input objects is given as a list of 2-element lists of the form [Bucket, Key] or 3-element lists of the form [Bucket, Key, KeyData].

You may also pass just the name of a bucket ({"inputs": "mybucket", ...}), which is equivalent to passing all of the keys in that bucket as inputs (i.e. "a map/reduce across the whole bucket"). You should be aware that this triggers the somewhat expensive "list keys" operation, so you should use it sparingly.

Query

The query is given as a list of phases, each phase being of the form {PhaseType: {...spec...}}. Valid PhaseType values are "map", "reduce", and "link".

Every phase spec may include a `keep` field, which must have a boolean value: `true` means that the results of this phase should be included in the final result of the map/reduce, `false` means the results of this phase should be used only by the next phase. Omitting the `keep` field accepts its default value, which is `false` for all phases except the final phase (Riak assumes that you were most interested in the results of the last phase of your map/reduce query).

Map

Map phases must be told where to find the code for the function to execute, and what language that function is in.

Function source can be specified directly in the query by using the "source" spec field. Function source can also be loaded from a pre-stored Riak object by providing "bucket" and "key" fields in the spec.

For example:

```
{"map": {"language": "javascript", "source": "function(v) { return [v]; }", "keep": true}}
```

would run the Javascript function given in the spec, and include the results in the final output of the m/r query.

```
{"map": {"language": "javascript", "bucket": "myjs", "key": "mymap", "keep": false}}
```

would run the Javascript function declared in the content of the Riak object under `mymap` in the `myjs` bucket, and the results of the function would not be included in the final output of the m/r query.

```
{"map": {"language": "erlang", "module": "riak_mapreduce", "function": "map_object_value"}}
```

would run the Erlang function `riak_mapreduce:map_object_value/3`.

Map phases may also be passed static arguments by using the "arg" spec field.

Reduce

Reduce phases look exactly like map phases, but are labeled "reduce".

Link

Link phases accept `bucket` and `tag` fields that specify which links match the link query. The string "" (underscore) in each field means "match all", while any other string means "match exactly this string". If either field is left out, it is considered to be set to "" (match all).

For example:





```
{"link":{"bucket":"foo","keep":false}}
```

would follow all links pointing to objects in the `foo` bucket, regardless of their tag.

MapReduce Screencast

With the syntax and query design fresh in your mind, take a few minutes to watch this screencast and check out Riak's MapReduce in action.

Here are some of the jobs we submitted in the screencast:

Name	Size	Creator	Creation Date	Comment
 simple-map.json	0.2 kB	Sean Cribbs	Jun 07, 2010 18:59	A simple map-only job that returns the entire data set.
 map-high.json	0.4 kB	Sean Cribbs	Jun 07, 2010 19:02	A map-reduce job that returns the maximum high sell value in the first week of January.
 map-highs-by-month.json	0.6 kB	Sean Cribbs	Jun 07, 2010 19:14	A more complicated map-reduce job that collects the max high by month.
 first-week.json	0.3 kB	Sean Cribbs	Jun 07, 2010 18:59	A simple map-only job that returns the values for the first week of January 2010.

Sample Functions

So you've seen us run some MapReduce jobs. Now it's time to try your hand at it.

Based on the sample data we loaded in the last section, here are some functions that should work for you. Take a few minutes to run them and, if you're feeling daring, modify them based on what you know about MapReduce in Riak to see if you can manipulate the results.



Submitting MapReduce queries from the shell

To run a query from the shell, here's the curl command to use:

```
curl -X POST http://127.0.0.1:8091/mapred -H "Content-Type: application/json" -d @-
```

After pressing return, paste your job in and then press Ctrl-D to submit it.

Map: find the days where the high was over \$600.00

Phase Function


```
function(value, keyData, arg) {  
  var data = Riak.mapValuesJson(value)[0];  
  if(data.High && data.High > 600.00)  
    return [value.key];  
  else  
    return [];  
}
```

Complete Job

```

{"inputs": "goog",
 "query": [{"map": {"language": "javascript",
                    "source": "function(value, keyData, arg) { var data =
Riak.mapValuesJson(value)[0]; if(data.High && parseFloat(data.High) > 600.00) return
[value.key]; else return [];}",
                    "keep": true}}]}

```

Name	Size	Creator	Creation Date	Comment
 sample-highs-over-600.json	0.3 kB	Sean Cribbs	Jun 07, 2010 19:20	

Map: find the days where the close is lower than open

Phase Function

```

function(value, keyData, arg) {
  var data = Riak.mapValuesJson(value)[0];
  if(data.Close < data.Open)
    return [value.key];
  else
    return [];
}


```

Complete Job

```

{"inputs": "goog",
 "query": [{"map": {"language": "javascript",
                    "source": "function(value, keyData, arg) { var data =
Riak.mapValuesJson(value)[0]; if(data.Close < data.Open) return [value.key]; else return [];}",
                    "keep": true}}]}

```

Name	Size	Creator	Creation Date	Comment
 sample-close-lt-open.json	0.3 kB	Sean Cribbs	Jun 07, 2010 19:20	

Map and Reduce: find the maximum daily variance in price by month

Phase functions

```

/* Map function to compute the daily variance and key it by the month */
function(value, keyData, arg){
  var data = Riak.mapValuesJson(value)[0];
  var month = value.key.split('-').slice(0,2).join('-');
  var obj = {};
  obj[month] = data.High - data.Low;
  return [ obj ];
}

/* Reduce function to find the maximum variance per month */
function(values, arg){
  return [ values.reduce(function(acc, item){
    for(var month in item){
      if(acc[month]) { acc[month] = (acc[month] < item[month]) ? item[month] :
acc[month]; }
      else { acc[month] = item[month]; }
    }
    return acc;
  })
];
}

```

Complete Job

```

{"inputs":"goog",
 "query":[{"map":{"language":"javascript",
 "source":"function(value, keyData, arg){ var data =
Riak.mapValuesJson(value)[0]; var month = value.key.split('-').slice(0,2).join('-'); var obj =
{}; obj[month] = data.High - data.Low; return [ obj ];}"}},
 {"reduce":{"language":"javascript",
 "source":"function(values, arg){ return [ values.reduce(function(acc, item){
for(var month in item){ if(acc[month]) { acc[month] = (acc[month] < item[month]) ? item[month] :
acc[month]; } else { acc[month] = item[month]; } } return acc; } ) ];}"}},
 "keep":true}}
]
}

```

Name	Size	Creator	Creation Date	Comment
 sample-max-variance-by-month.json	0.6 kB	Sean Cribbs	Jun 07, 2010 19:20	

A MapReduce Challenge

Here is a scenario involving the data you already have loaded up. If you have a moment, try to solve it using what you've just learned about MapReduce:



MapReduce Challenge

Find the largest day for each month in terms of dollars traded, and subsequently the largest overall day.

Hint: You will need at least one each of map and reduce phases.

What's Next? ----> You've done some basic MapReduce queries in a distributed key/value store. For the last leg of the Fast Track, let's take a look at the tunable CAP controls in Riak.



Additional Reading for this Section

- [MapReduce In Depth](#)
- [Erlang MapReduce](#)
- [A Longer MapReduce Screencast](#)
- [A list of prebuilt JavaScript MapReduce Functions that ship with Riak](#)
- [Google's Original MapReduce Paper](#)

Tunable CAP Controls in Riak

So, we've come a long way. If you've done the Fast Track in order, that means you've had a short intro to Riak, set up a three node cluster on your local machine, worked a bit with the HTTP interface, and performed some MapReduce queries.

In the last section of the Fast Track, we are going to talk about how Riak distributes your data around the cluster and lets you tune your levels of consistency and availability. This has immense value and implications for your applications, and it's one of the Riak features that we feel truly differentiates us.

At the bottom of this page there is a final screencast that briefly touches on how to adjust your replication levels to match you application and business needs. Before you watch that, however, have a quick read of the content below.

A Primer on N, R, and W

Riak exposes "CAP Controls" to the developers in such a way that they can, down to the Bucket level, tune how many copies of data we want to store. We do this using N, R, and W values.

Riak's guiding design principle is Dr. Eric Brewer's CAP Theorem. The CAP theorem defines distributed systems in terms of three desired properties: Consistency, Availability, and Partition (failure) tolerance. The theorem states you can only rely on having two of the three properties at any time.

Riak chooses to focus on the A and P of CAP. The choice puts Riak in the eventually consistent camp. However, the window for "eventually consistent" is in terms of milliseconds which can be good enough for many applications.

N Value and Replication

All data stored in Riak will be replicated to a number of nodes in the cluster according to the N value (`n_val`) property set on the bucket. By default, Riak chooses an `n_val` of 3 for you. This means that data stored in the bucket will be replicated to 3 different nodes, thus storing three copies. For this to be effective, you need at least three physical nodes in your cluster. (We can, however, demonstrate its merits with three local nodes.)

To change the N value for a bucket (to something different than the default) issue a PUT request to the bucket with the new N value. If you still have your three node Riak cluster running, try this:

```
$ curl -v -X PUT -H "Content-Type: application/json" -d '{"props":{"n_val":2}}' http://127.0.0.1:8091/riak/another_bucket
```

This will change the `n_val` of the bucket "another_bucket" to two, meaning that each piece of data in that bucket will be replicated to two partitions in the cluster.



A Word on Setting the N Value

`n_val` must be greater than 0 and less than or equal to the number of actual nodes in your cluster to get all the benefits of replication. And, we advise against modifying the `n_val` of a bucket after its initial creation as this may result in failed reads because the new value may not be replicated to all the appropriate partitions.

R Value and Read Failure Tolerance

So we changed the Bucket `n_val` to 2 with that last command.

Riak forces the client to supply a "R value" on each direct fetch. The R value represents the number of Riak nodes which must return results for a read before the read is considered successful. This allows Riak to provide read availability even when nodes are down or laggy.

For example, in this HTTP request, the `r` value is set to 1:

```
http://127.0.0.1:8091/riak/images/1.png?r=1
```


This means that Riak will only return a copy of that data if at least 1 copy is present in your cluster.

W Value and Write Fault Tolerance

Riak's API forces the client to supply a "W value" on each update. The W value represents the number of Riak nodes which must report success before an update is considered complete. This allows Riak to provide write availability even when nodes are down or laggy.

In this PUT operation, you can see the w value set to 3.

```
$ curl -v -X PUT http://127.0.0.1:8091/riak/docs/story.txt?w=3 -H "Content-type: text/plain"
--data-binary @story.txt
```

N, R and W in Action

Here is brief screencast that will show you just how the N, R, and W values function in our running three node Riak cluster:

What's next?

Congratulations. If you're reading this and have done the Fast Track in order, it means that you've built a three node Riak cluster, inserted a small amount of data with the help of some scripts, performed basic API operations, queried that data with MapReduce, and have an introduction to the powers of tunable CAP controls. Needless to say, you've come a long way.

But, there is always more Riak learning to be done, so here is a list a next steps you may want to take if you are still interested in Riak. Whatever you do, if you have a moment, send an email to mark@basho.com with your thoughts on what you thought we did well, and, more importantly, how we can make the Riak Fast Track better.

- If you'd like more general information about Riak, check out [Recommended Resources](#)
- For more info on how to interact with Riak from your language of choice, start with [Client Libraries](#)
- To download a platform-specific package of Riak, check out [Getting Started](#)
- If you're interested in knowing how Riak stacks up to a few other databases, have a look at [Riak Comparisons](#)
- If you want to have a look at the Riak Source you can do so on [GitHub](#) or [Bitbucket](#)

Thank You

The [Riak Fast Track](#) is constantly improving thanks to the help of all who review it and send their feedback. This is a list of those people.

So, in no particular order, **Thank You!**

- Michael Guterl
- Andrew Tunnell-Jones
- Bruno Michel
- François Beausoleil
- Kirk Bateman
- Norman Khine
- Matthew Pflueger
- Alexander Sicular
- Maxime Garcia
- Harmen Wessels
- Colin Clark
- Sean Murphy
- Dirceu Pereira Tiegs
- Thilo
- Tyler Weir
- Dick Davies
- Seth Falcon
- Ray Cote
- Seth Edwards
- Rick Thomas
- Seth Falcon
- Eric Cestari
- Wilson MacGyver
- Jacob Swanner

Installation and Setup

Installation

Click the link for your OS below to see instructions on how to install Riak:

- [Installing on Debian and Ubuntu](#)
- [Installing on RHEL and CentOS](#)
- [Installing on Mac OS X](#)
- [Installing from source](#)

Starting up

To start up a Riak node, change directory as necessary to where you installed Riak (in the source directory, it's `rel/riak/bin`) and run the `riak` command like so:

```
$ riak start
```

To run Riak with an interactive Erlang console:

```
$ riak console
```

Once your node has started, you can double-check that it is running using:

```
$ riak ping
pong
```

The command will respond with `pong` if the node is running, or `pang` if it is not.

Does it work?

The easiest way to test whether Riak is working is to use the `curl` command-line tool. Now that you have Riak running on your local machine, run this command:

```
$ curl -v http://127.0.0.1:8098/riak/test
```

You should get a response that looks like this:

```
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 200 OK
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Mon, 08 Mar 2010 16:42:49 GMT
< Content-Type: application/json
< Content-Length: 266
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"props":{"name":"test","allow_mult":false,"big_vclock":50,"chash_keyfun":{"mod":"riak_util",
"fun":"chash_std_keyfun"},"linkfun":{"mod":"raw_link_walker_resource","fun":"mapreduce_linkfun"
},"n_val":3,"old_vclock":86400,"small_vclock":10,"young_vclock":20},"keys":[]}
```

Now what?

You have a working Riak system! From here you might want to check out:

- [Client Libraries for connecting Riak to your favorite programming language](#)
- [Add more nodes to your Riak cluster](#)
- [The high level overview of the Riak design and functionality](#)
- [The more in depth look at how Riak works.](#)

Installing Erlang

Riak requires [Erlang R13B03](#) or later (R13B04 for Riak 0.10 and later). For Erlang to build and install, you must have a GNU-compatible build system, and the development bindings of `ncurses` and `openssl`.



The Riak binary packages for [Debian and Ubuntu](#), [Mac OS X](#) and [RHEL and CentOS](#) do not require that you build Erlang from source. *You will have to download and install Erlang, however, if you're planning on completing [The Riak Fast Track](#)*

- [Installing on GNU/Linux](#)
- [Installing on Mac OS/X](#)
 - [Source](#)
 - [Homebrew](#)
 - [MacPorts](#)

Installing on GNU/Linux

Most distributions *do not* have the most recent Erlang release available, **so you will need to install from source**.

First, make sure you have a compatible build system and the `ncurses` and `openssl` development libraries installed. On Debian/Ubuntu use this command:

```
$ sudo apt-get install build-essential libncurses5-dev openssl libssl-dev
```

On RHEL/CentOS use this command:

```
$ sudo yum install gcc glibc-devel make ncurses-devel openssl-devel
```

Next, download, build and install Erlang:

```
$ wget http://erlang.org/download/otp_src_R13B04.tar.gz
$ tar zxvf otp_src_R13B04.tar.gz
$ cd otp_src_R13B04
$ ./configure && make && sudo make install
```

Installing on Mac OS/X

You can install Erlang in several ways on OS/X, from [source](#), with [Homebrew](#) or [MacPorts](#).

Source

To build from source, you must have XCode tools installed from the CD that came with your Mac or from Apple's [Developer website](#).

First, download and unpack the source:

```
$ curl -O http://erlang.org/download/otp_src_R13B04.tar.gz
$ tar zxvf otp_src_R13B04.tar.gz
$ cd otp_src_R13B04
```

Next, configure Erlang. If you're on Snow Leopard (OS/X 10.6) or Leopard (OS/X 10.5) with an Intel processor:

```
$ ./configure --enable-hipe --enable-smp-support --enable-threads \
--enable-kernel-poll --enable-darwin-64bit
```

If you're on a non-Intel processor or older version of OS/X:

```
$ ./configure --enable-hipe --enable-smp-support --enable-threads --enable-kernel-poll
```

Now build and install:

```
$ make && sudo make install
```

You will be prompted for your sudo password.

Homebrew

If you intend to [install Riak with Homebrew](#), simply follow those instructions and Erlang will be installed automatically. To install it separately:

```
$ brew install erlang
```

MacPorts

Installing with MacPorts is easy:

```
$ port install erlang
```

Installing on Debian and Ubuntu

The following steps should get you up and running with Riak on Debian or Ubuntu 9.10. You can install from source or from our custom .deb package.

From our custom .deb package

To install from our pre-built package, run these commands for the appropriate platform:

64-bit

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak_0.11.0-1344_amd64.deb
$ sudo dpkg -i riak_0.11.0-1344_amd64.deb
```

32-bit

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak_0.11.0-1344_i386.deb
$ sudo dpkg -i riak_0.11.0-1344_i386.deb
```

From source

Riak requires [Erlang R13B04](#) or later. If you do not have Erlang already installed, see [Installing Erlang](#). Don't worry, it's easy!

Let's install the dependencies that we can from apt:

```
$ sudo apt-get install build-essential libc6-dev-i386
```

If you do not have Erlang installed, please [do so](#).

Now we can download and install Riak:

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak-0.11.0.tar.gz
$ tar zxvf riak-0.11.0.tar.gz
$ cd riak
$ make rel
```

There should now be a fresh build of Riak in the rel/riak directory.

Next Steps: [Basic Cluster Setup](#) will show you how to go from one node to bigger than Google

Installing on Mac OS X

The following steps should get you up and running with Riak on MacOSX 10.5 or 10.6. You can install from source or download a precompiled tarball.



ulimit on OS X

OS X gives you a very small limit on open file handles, so even with a backend that uses very few file handles, it's possible to run out. Run this command before getting started developing on your Mac:

```
ulimit -n 512
```

From precompiled tarballs

To run Riak from our precompiled tarball, run these commands for the appropriate platform:

64-bit

```
$ curl -O http://downloads.basho.com/riak/riak-0.11/riak-0.11.0-osx-x86_64.tar.gz
$ tar xzvf riak-0.11.0-osx-x86_64.tar.gz
```

32-bit

```
$ curl -O http://downloads.basho.com/riak/riak-0.11/riak-0.11.0-osx-i386.tar.gz
$ tar xzvf riak-0.11.0-osx-i386.tar.gz
```

After the release is untared you will be able to cd into the riak directory and execute bin/riak start to start the Riak node.

Homebrew

Installing with Homebrew is easy:

```
$ brew install riak
```

Homebrew will install Erlang if you don't have it already.

From source

You must have XCode tools installed from the CD that came with your Mac or from Apple's [Developer website](#).

Riak requires [Erlang R13B04](#) or later. If you do not have Erlang already installed, see [Installing Erlang](#). Don't worry, it's easy!

Next, download and unpack the source distribution.

```
$ curl -O http://downloads.basho.com/riak/riak-0.11/riak-0.11.0.tar.gz
$ tar zxvf riak-0.11.0.tar.gz
$ make rel
```

If you get errors when building about "incompatible architecture", please verify that you built Erlang with the same architecture as your system (Snow Leopard - 64bit, everything else - 32bit).

Next Steps: [Basic Cluster Setup](#) will show you how to go from one node to bigger than Google

Installing on RHEL and CentOS

The following steps should get you up and running with Riak on Centos or Redhat. You can install from source or from our custom .rpm package. Please note, CentOS enables SE Linux by default and you may need to disable SE Linux if you encounter errors.

From our custom .rpm package

To install from our pre-built package, run these commands for the appropriate platform:

64-bit

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak-0.11.0-1344.e15.x86_64.rpm
$ sudo rpm -Uvh riak-0.11.0-1344.e15.x86_64.rpm
```

32-bit

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak-0.11.0-1344.e15.i386.rpm
$ sudo rpm -Uvh riak-0.11.0-1344.e15.i386.rpm
```

From source

Riak requires [Erlang R13B04](#) or later. If you do not have Erlang already installed, see [Installing Erlang](#). Don't worry, it's easy!

Building from source will require the following packages:

```
gcc
glibc-devel
make
```

You can install these with yum:

```
$ sudo yum install gcc glibc-devel make
```

If you do not have Erlang installed, please [do so](#).

Now we can download and install Riak:

```
$ wget http://downloads.basho.com/riak/riak-0.11/riak-0.11.0.tar.gz
$ tar zxvf riak-0.11.0.tar.gz
$ cd riak
$ make rel
```

There should now be a fresh build of Riak in the rel/riak directory.

Next Steps: [Basic Cluster Setup](#) will show you how to go from one node to bigger than Yahoo!

Installing Riak from Source

Riak should be installed from source if you are building on a platform for which a package does not exist or you are interested in contributing to Riak.

Dependencies

Riak requires [Erlang](#) R13B04 or later. If you do not have Erlang already installed, see [Installing Erlang](#). Don't worry, it's easy!

Installation

The following instructions generate a complete, self-contained build of Riak in `$(RIAK)/rel/riak` where `$(RIAK)` is the location of the unpacked or cloned source.

Installing from source package

Download the Riak source package from the [Download Center](#) and build:

```
$ curl -O http://downloads.basho.com/riak/riak-0.11/riak-0.11.0.tar.gz
$ tar zxvf riak-0.11.0.tar.gz
$ cd riak-0.11.0
$ make rel
```

Installing from Bitbucket

Clone the repository using [Mercurial](#) and build:

```
$ hg clone http://hg.basho.com/riak
$ cd riak
$ make rel
```

Platform Specific Instructions

For instructions about specific platforms, see:

- [Installing on Debian and Ubuntu](#)
- [Installing on Mac OS X](#)
- [Installing on RHEL and CentOS](#)

If you are running Riak on a platform not in the list above, let us know by sending an email to riak@basho.com.

Windows

Riak is not currently supported on Microsoft Windows.

Installing with Chef

If you manage your infrastructure with Chef it is possible to install Riak with the use of the cookbook located in the Opscode cookbook repository. You can find the cookbook as well as usage information [here](#).

The cookbook will allow you run a single node, but running a cluster will require customization of the appropriate configuration files. You can find more information on that subject [here](#) and in the [The Riak Fast Track](#).

Basic Cluster Setup

The configuration of a Riak cluster requires configuring a node to listen on a non-local interface (i.e., not 127.0.0.1), and then joining nodes together.

Begin by editing your `etc/app.config` file. The `app.config` file will be located in your `rel/riak/etc/` directory if you compiled from source, and `/etc/riak/` if you used a binary install of Riak.

The commands below assume you are running from a source install, but if you have installed Riak with one of our binary installs you can substitute the usage of `bin/riak` with `sudo /usr/sbin/riak` and `bin/riak-admin` with `sudo /usr/sbin/riak-admin`.

- [Configure the First Node](#)
- [Add a Second Node to Your Cluster](#)
- [Running Multiple Nodes on One Host](#)
- [Multiple Clusters on One Host](#)

Configure the First Node

First, stop your Riak node if it is currently running:

```
$ bin/riak stop
```

Change the following line in your `etc/app.config` to have the node listen on the network interface you want your cluster to operate on, such as 192.168.1.10:

```
{riak_web_ip, "127.0.0.1"},
```

becomes

```
{riak_web_ip, "192.168.1.10"},
```

Next edit the `etc/vm.args` file and change the `-name` to your new IP:

```
-name riak@127.0.0.1
```

becomes

```
-name riak@192.168.1.10
```

Start the Riak node:

```
$ bin/riak start
```

If you have started this Riak node before, you will need to "reip" the system so the node can take appropriate measures to update its ring file.

```
$ bin/riak-admin reip riak@127.0.0.1 riak@192.168.1.10
```

This node is now configured and ready to have a node join it in the Riak cluster.

Add a Second Node to Your Cluster

Repeat the steps above for another other host on the network. Once the node has started you will use the `bin/riak-admin` command to have it join the other node in the Riak cluster.

```
$ bin/riak-admin join riak@192.168.1.10
#Sent join request to riak@192.168.1.10
```

Your second Riak node is now part of the cluster and as begun syncing with your first node. There are two ways to see if your Riak cluster has been correctly configured.

Using riak-admin:

```
$ bin/riak-admin status | grep ring_members
#ring_members : ['riak@192.168.1.10', 'riak@192.168.1.11']
```

Using riak attach:

```
$ riak attach
1> {ok, R} = riak_ring_manager:get_my_ring().
{ok, {chstate, 'riak@192.168.1.10', .....
(riak@192.168.52.129)2> riak_ring:all_members(R).
['riak@192.168.1.10', 'riak@192.168.1.11']
```

You can repeat the above steps to add more nodes to your cluster.

Running Multiple Nodes on One Host

If you have built Riak from source code, or if you are using the Mac OSX pre-built package, then you can easily run multiple Riak nodes on the same machine. The most common scenario for doing this is to experiment with running a Riak cluster. (Note: if you have installed the .deb or .rpm package, then you will need to download and build Riak source to follow the directions below.)

To run multiple nodes, make copies of the `riak` directory.

- If you have run `make all rel`, then this can be found in `./rel/riak` under the root Riak source directory.
- If you are running Mac OSX, then this is the directory where you unzipped the .tar.gz file.

Now, assuming that you made copied `./rel/riak` into `./rel/riak1`, `./rel/riak2`, `./rel/riak3`, etc.:

- In `riakN/etc/app.config`, change the `web_port`, `handoff_port`, and `pb_port` to unique ports for each node..
- In `riakN/etc/vm.args`, change the line that says `-name riak@127.0.0.1` to a unique name for each node, for example: `-name riak1@127.0.0.1`.

Now, start the nodes, changing path names and nodes as appropriate:

```
./rel/riak1/bin/riak start
./rel/riak2/bin/riak start
./rel/riak3/bin/riak start
(etc.)
```

Next, join the nodes into a cluster:

```
./rel/riak2/bin/riak-admin join riak1@127.0.0.1
./rel/riak3/bin/riak-admin join riak1@127.0.0.1
(etc.)
```

Alternatively, you can run `make all devrel`, which will create three copies of Riak under the directories `./dev/dev1`, `./dev/dev2`, and `./dev/dev3`, with the configuration pre-set to allow you to run the nodes simultaneously. (Note: The Web port for the three nodes is 8091, 8092, and 8093, which is different from the default of 8098 that you get when you run "make all rel".)

You can start each of them and join them together using the commands shown above, substituting directory names as appropriate.

Multiple Clusters on One Host

Using the above technique it is possible to run multiple clusters on one node. If a node hasn't joined an existing cluster it will behave just as a cluster would. Running multiple clusters on one node is simply a matter of having two or more distinct nodes or groups of nodes joined together.

Setting Up Innostore

Setting Up Innostore

Innostore is distributed separately from Riak and will need to be compiled prior to installing. You will need to have [Erlang installed](#) to compile Innostore.

Downloading & Installing Innostore on Debian/Ubuntu

64-bit


```
$ wget http://downloads.basho.com/innostore/innostore-1.0.0/innostore_1.0.0-88_amd64.deb
$ sudo dpkg -i innostore_10-1_amd64.deb
```

32-bit

```
$ wget http://downloads.basho.com/innostore/innostore-1.0.0/innostore_1.0.0-88_i386.deb
$ sudo dpkg -i innostore_10-1_i386.deb
```

Downloading & Installing Innostore on RHEL/CentOS

64-bit

```
$ wget http://downloads.basho.com/innostore/innostore-1.0.0/innostore-1.0.0-88.el5.x86_64.rpm
$ sudo rpm -Uvh innostore-10-1.x86_64.rpm
```

32-bit

```
$ wget http://downloads.basho.com/innostore/innostore-1.0.0/innostore-1.0.0-88.fc12.x86_64.rpm
$ sudo rpm -Uvh innostore-10-1.i386.rpm
```

Downloading & Installing Innostore from Source

You can download Innostore [here](#). A simple way to do this would be using wget:

```
$ wget http://downloads.basho.com/innostore/innostore-1.0.0/innostore-1.0.0.tar.gz
```

Change to the Innostore directory and compile:



Single CPU Hosts

You will need to run the following to allow Innostore to compile:

```
$ export ERL_FLAGS="-smp enable"
```

Untar and compile Innostore:

```
$ tar xzvf innostore-1.0.0.tar.gz
$ cd innostore-1.0.0
$ make
```

If Innostore compiled as expected you should see the following:

```
./rebar compile eunit verbose=1
.....
.....
=====
All 7 tests passed.
Cover analysis: /home/XXXXX/innostore/.eunit/index.html
```

If your compile passed tests you are now ready to install Innostore into your Riak distribution. If you are using a copy of Riak you compiled yourself you can install Innostore by issuing the following command replacing \$RIAK with the location of your Riak install:

```
$ ./rebar install target=$RIAK/lib
```

If you installed Riak using one of our binary packages you would use the following command:

```
$ ./rebar install target=/usr/lib/riak
```

Configuring Innostore

The last step in setting up Innostore is to configure the Riak node to use Innostore as its storage backend by editing your `app.config` file located in your Riak installs `$RIAK/etc/` directory or `/etc/riak/app.config`. If your Riak node is running issue it the stop command.

First change the configured storage engine to use Innostore:

```
{storage_backend, riak_kv_dets_backend},
```

becomes

```
{storage_backend, riak_kv_innystore_backend},
```

You will also need to append the Innostore configuration to the end of your `app.config` file. A good place for this is after the SASL section (you will need to add a comma after the existing SASL section):

```
#Binary Install
{sasl, [ ....
]}, %% < -- make sure you add a comma here!!

{innystore, [
  {data_home_dir, "/var/lib/riak/innodb"}, %% Where data files go
  {log_group_home_dir, "/var/lib/riak/innodb"}, %% Where log files go
  {buffer_pool_size, 268435456} %% 256MB in-memory buffer in bytes
]}

#Source Install
{sasl, [ ....
]}, %% < -- make sure you add a comma here!!

{innystore, [
  {data_home_dir, "$RIAK/data/innodb"}, %% Where data files go
  {log_group_home_dir, "$RIAK/data/innodb"}, %% Where log files go
  {buffer_pool_size, 268435456} %% 256MB in-memory buffer in bytes
]}
}
```



Single CPU Hosts

You will need to add the following to Riak's `vm.args` file:

```
-smp enable
```

Innystore will either fail to function or function erratically without the flag.

This is set by default in 0.10 and later.



RHEL/CentOS

You will need to add the following to Riak's `vm.args` file:

```
-pa /usr/lib64/innystore/ebin
```

Riak will fail to find the modules to use innostore otherwise. Riak will fail to startup with an `invalid_storage_backend` error.

Now that you have configured Innostore you can test your install by running to following command:

```
# Binary Install
$ sudo /usr/sbin/riak console

# Source Install
$ $RIAK/bin/riak console
```

If the install was successful you will see something similar to:

```
100220 16:36:58 InnoDB: highest supported file format is Barracuda.
100220 16:36:58 Embedded InnoDB 1.0.3.5325 started; log sequence number 45764
```

Once you have confirmed your install is successful exit the console and start your Riak node with the start command:

```
(riak@127.0.0.1)1> q()
```

An Introduction to Riak

This section is a high level overview of concepts, technology choices, and implementation details that are at work in Riak.

- [Basics](#)
- [Data Storage](#)
- [Clustering](#)
- [Reading Data](#)
- [Writing and Updating Data](#)
- [Querying and MapReduce](#)
- [The Riak API](#)

Basics

Languages Used

Written in [Erlang](#) and C primarily with a small amount of Javascript

Clients

Basho supported Drivers are available for Erlang, Python, Java, PHP, Javascript, and Ruby.

- [Take a more in depth look at the Client Libraries](#)

History

Riak is based on technology originally developed by Basho Technologies to run a Salesforce automation business. There was more interest in the datastore technology than the applications built on it so Basho decided to build a business around Riak itself.

- [Learn more about Basho Technologies, the company behind Riak](#)

Influences

Riak is heavily influenced by Dr. Eric Brewer's CAP Theorem and Amazon's Dynamo paper. Most of the core team comes from Akamai which also explains their focus on operational ease and fault tolerance.

Enterprise vs. Open Source

Riak comes in two flavors: open source and enterprise. Enterprise is a superset of the open source version with a few features added. The current enterprise features are SNMP support, inter-datacenter replication (not to be confused with *intra-datacenter* replication, which is supported in the open source version), web-based administration interface, and top-tier support.

- [Learn more about Riak EnterpriseDS](#)

Data Storage

Buckets and keys

Buckets and keys are the only way to organize data inside of Riak. User data is stored and referenced by bucket/key pairs.

Links and Metadata

Bucket/key entries, hereafter referred to as Riak objects, can store links pointing to other entries. These links can be walked via Riak's HTTP interface directly or as part of a map/reduce job. Riak objects and buckets can store user-defined metadata in addition to the actual data payload. The metadata is exposed as HTTP headers in the HTTP interface and as associative arrays inside map/reduce jobs.

Pluggable Backends

Riak uses an API to interact with its storage subsystem. The API allows Riak to support multiple backends which can be "plugged in" as needed. Riak currently ships with backends for dets, ets, Erlang's balanced trees (gb_trees), and writing directly to the filesystem. Riak also supports per-bucket backends.

Other Backends

Basho also develops a Riak backend named Innostore based on the embeddable version of InnoDB. It is reputed to be the fastest backend currently available. Due to licensing restrictions, Innostore is provided separately.

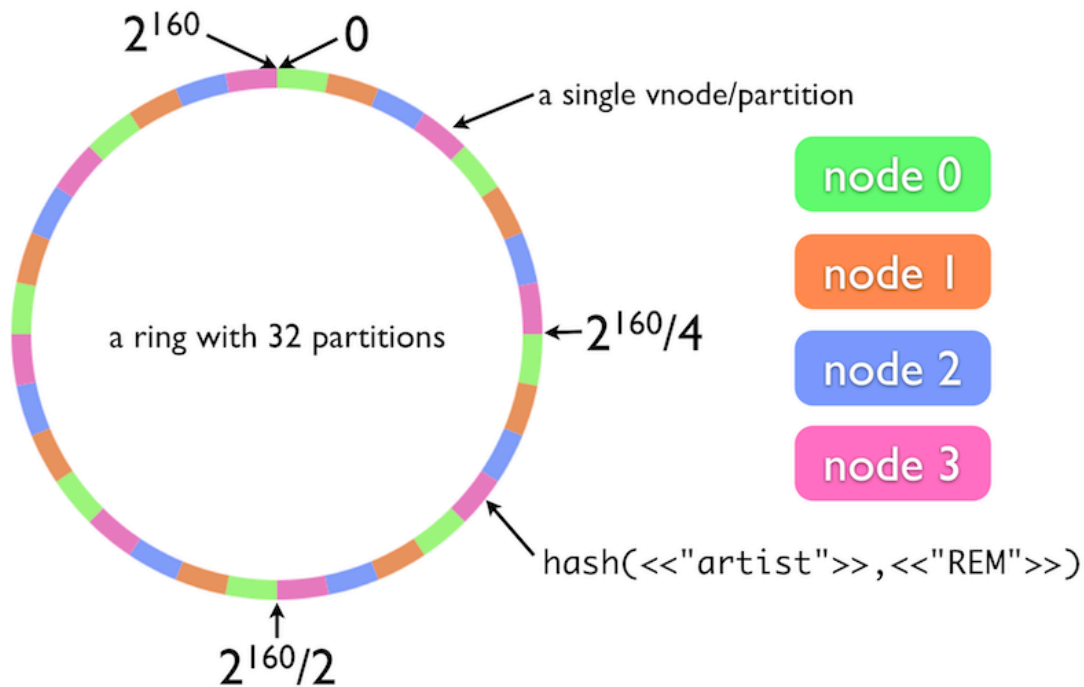
Clustering

Nodes, Vnodes, and Partitions

Central to any Riak cluster is a 160-bit integer space which is divided into equally-sized partitions.

Physical servers, referred to in the cluster as "nodes," run a certain number of virtual nodes, or "vnodes". Each vnode will claim a partition on the ring. The number of active vnodes is determined by the number of physical nodes in the a cluster at any given time.

As a rule, each node in the cluster is responsible for $1/(\text{total number of physical nodes})$ of the ring. You can determine the number of vnodes on each node by calculating $(\text{number of partitions})/(\text{number of nodes})$. More simply put, a ring with 32 partitions, composed of four physical nodes, will have approximately eight vnodes per node. This setup is represented in the diagram below.



Nodes can be added and removed from the cluster dynamically and Riak will redistribute the data accordingly.

Distribution

Riak is designed, from the ground up, to run in a distributed environment. Core operations, such as read/writing data and executing map/reduce jobs, actually become faster when more Riak nodes are added to a cluster.

Riak & CAP Theorem

Riak's guiding design principle is Dr. Eric Brewer's CAP Theorem. The CAP theorem defines distributed systems in terms of three desired properties: Consistency, Availability, and Partition (failure) tolerance. The theorem states you can only rely on having two of the three properties at any time.

Riak chooses to focus on the A and P of CAP. The choice puts Riak in the eventually consistent camp. However, the window for "eventually consistent" is in terms of milliseconds which can be good enough for many applications.

No master node

All nodes in a Riak cluster are equal. Each node is fully capable of serving any client request. This is possible due to the way Riak uses consistent hashing to distribute data around the cluster.

Storage implications

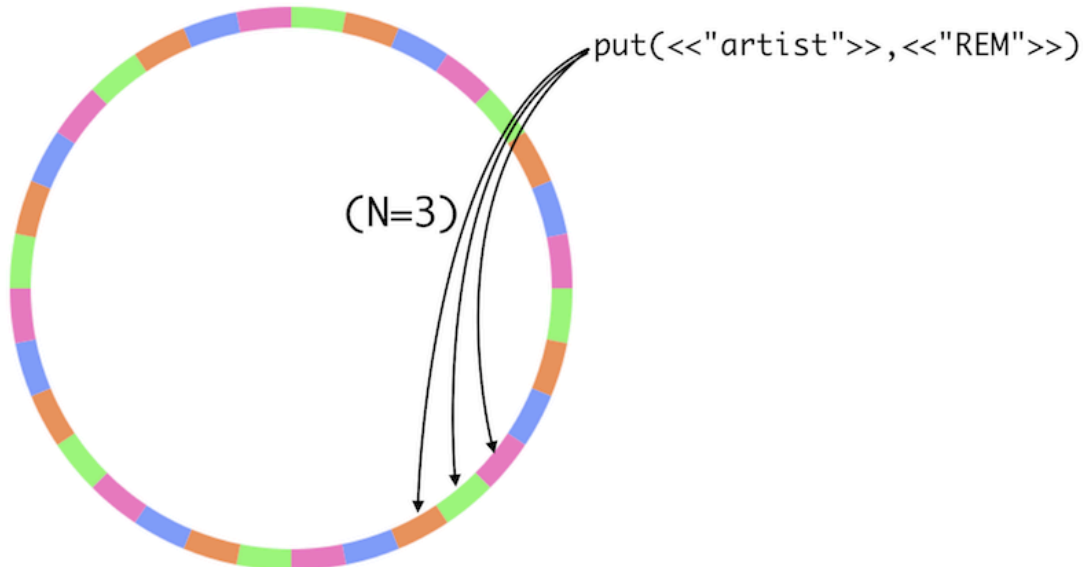
Riak communicates bucket information around the cluster using a [gossip protocol](#). In general, large numbers of buckets within a Riak cluster is not a problem. In practice, there are two potential restrictions on the maximum number of buckets a Riak cluster can handle.

First, buckets which use a non-standard set of properties will force Riak to gossip more data around the cluster. The additional data can slow processing and place an upper limit on performance. Second, some backends, namely Innostore, store each bucket as a separate entity. This can cause a node to run out of resources such as file handles. These resource restrictions might not impact performance but they can represent another limit on the maximum number of buckets.

Replication

Riak controls how many replicas of data are kept via a setting called the "N value". This value has a per-node default but can be overridden on each bucket. Riak objects inherit the N value of their parent bucket. All nodes in the same cluster should agree on and use the same N value.

For example, here is a depiction of what happens when `n_val = 3` (This is the default setting). When you store a datum in a bucket with an N value of three, the datum will be replicated to three separate partitions on the Riak Ring.



- [Configuring Replication in Riak](#)

Hinted Handoff

Riak uses a technique called 'hinted handoff' to compensate for failed nodes in a cluster. Neighbors of a failed node will pick up the slack and perform the work of the failed node allowing the cluster to continue processing as usual. This can be considered a form of self-healing.

Dynamic Growth

Riak will automatically re-balance data as nodes join and leave the cluster.

Reading Data

Fetching

Riak objects can be fetched directly if the client knows the bucket and key. This is the fastest way to get data out of Riak.

R Value

Riak forces the client to supply a "R value" on each direct fetch. The R value represents the number of Riak nodes which must return results for a read before the read is considered successful. This allows Riak to provide read availability even when nodes are down or laggy.

Read failure tolerance

Subtracting R from N will tell you the number of down or laggy nodes a Riak cluster can tolerate before becoming unavailable for reads. For example, an 8 node cluster with an N of 8 and a R of 1 will be able to tolerate up to 7 nodes being down before becoming unavailable for reads.

Link walking

Riak can also return objects based on links stored on the object. Link walking can be used to return a set of related objects from a single request.

Writing and Updating Data

Vector clocks

Each update to a Riak object is tracked by a vector clock. Vector clocks allow Riak to determine causal ordering and detect conflicts in a distributed system.

Conflict resolution

Riak has two ways of resolving update conflicts on Riak objects. Riak can allow the last update to automatically "win" or Riak can return both versions of the object to the client. This gives the client the opportunity to resolve the conflict on its own.

W Value

Riak's API forces the client to supply a "W value" on each update. The W value represents the number of Riak nodes which must report success before an update is considered complete. This allows Riak to provide write availability even when nodes are down or laggy.

Write failure tolerance

Subtracting W from N will tell you the number of down or laggy nodes a Riak cluster can tolerate before becoming unavailable for writes. For example, an 8 node cluster with an N of 8 and a W of 2 will be able to tolerate up to 6 nodes being down before becoming unavailable for writes.

Querying and MapReduce

Querying Riak is performed by map/reduce processing. Riak has a flexible map/reduce mechanism which more closely resembles Hadoop than CouchDB, for instance. Map/Reduce jobs are submitted via HTTP POST to a pre-defined URL which defaults to '/mapred'. Jobs are described in JSON using a set of nested hashes describing the inputs, phases, and timeout for a job. Data is returned in JSON-encoded form.

Phases

Map/Reduce jobs are chains of functions. Riak combines the function along with information about what to do with the output and the language the function is written into a 'phase'. The output of one phase becomes the input for the next. Riak can accumulate the output of any phase or set of phases and return the data to the client as part of that job's output.

Map Phase

Map phases are responsible for gathering data from Riak bucket/key entries. Map phases can operate on entire buckets or lists of bucket/key pairs. The fetching of data is highly parallel and scales with the number of nodes in the cluster. Riak caches the results of map fetches to reduce the load on the storage backends. This means subsequent map fetches for the same data will be faster after the initial fetch.

Link Phase

Link phases are a specialized version of map phases. A link phase fetches Riak objects based on a link walk. Link phases can be used to perform map/reduce processing on sets of related objects.

Reduce Phase

Reduce phases can perform arbitrary processing on the data retrieved from map or link phases. Unlike CouchDB, Riak reduce phases are not required to return a single answer.

Query Languages

Erlang

Map/reduce functions can be written in Erlang. Erlang function have complete access to the Riak Erlang API.

Javascript

Riak recently added support for writing map/reduce functions in Javascript. Mozilla's Spidermonkey engine provides the runtime environment. Pre-defined Javascript functions run almost as fast as Erlang functions. Javascript functions are currently permitted read-only access to Riak.

- [Watch a screencast overview of MapReduce in Riak](#)

The Riak API

Data Storage

The guys who wrote Riak are also responsible for the Erlang REST framework Webmachine, so it's not surprising Riak uses REST for its API. Store operations use HTTP PUTs or POSTs and fetches use HTTP GETs. Storage operations are submitted to a pre-defined URL which defaults to '/riak'.

Clients can set the Content-Type header for each Riak object. Riak will replay the header when the object is fetched. This is nice since it allows Riak to be content agnostic.

- [Take an in depth look at the Riak's REST API](#)

How Things Work

- [Summary](#)
- [The Ring](#)
- [Gossiping](#)
- [Vector clocks](#)
- [Backends](#)
- [More reading](#)

Summary

Riak is a distributed key-value store, strongly influenced by the [Dynamo Paper](#) and the [CAP Theorem](#). It supports high availability by allowing tunable levels of guarantees for durability and eventual consistency.

A Riak cluster is generally run on a set of well-connected physical hosts. Each host in the cluster runs one Riak node. Each Riak node runs a set of virtual nodes, or "vnodes", that are each responsible for storing a separate portion of the key space.

Nodes are not clones of each other, nor do they all participate in fulfilling every request. The extent to which data is replicated, and when, and with what merge strategy and failure model, is configurable at runtime.

The Ring

(Much of this section is discussed in the Dynamo paper, but it's a good summary of how Riak implements the necessities.)

Riak's client interface speaks of "buckets" and "keys". Internally, Riak computes a 160-bit binary hash of the bucket/key pair, and maps this value to a position on an ordered "ring" of all such values. This ring is divided into partitions. Each Riak vnode is responsible for a partition (we say that it "claims" that partition).

The nodes of a Riak cluster each attempt to run roughly an equal number of vnodes. In the general case, this means that each node in the cluster is responsible for $1/(\text{number of nodes})$ of the ring, or $(\text{number of partitions})/(\text{number of nodes})$ vnodes. For example, if two nodes define a 16-partition cluster, then each node will run 8 vnodes. Nodes claim their partitions at random intervals around the ring, in an attempt at even distribution.

When a value is being stored in the cluster, any node may participate as the coordinator for the request. The coordinating node consults the ring state to determine which vnode owns the partition in which the value's key belongs, then sends the "put" request to that vnode, as well as the vnodes responsible for the next $N-1$ partitions in the ring, where N is a bucket-configurable parameter that describes how many copies of the value to store. The put request may also specify that at least W ($= < N$) of those vnodes reply with success, and that DW ($= < W$) reply with success only after durably storing the value.

A fetch, or "get", request operates similarly, sending requests to the vnode that "claims" the partition in which the key resides, as well as to the next $N-1$ partitions. The request also specifies R ($= < N$), the number of vnodes that must reply before a response is returned.

Gossiping

The ring state is shared around the cluster by means of a "gossip protocol". Whenever a node changes its claim on the ring, it announces its change via this protocol. It also periodically re-announces what it knows about the ring, in case any nodes missed previous updates.

Vector clocks

With any node able to drive any request, and not all nodes needing to participate in each request, it is necessary to have a method for keeping track of which version of a value is current. This is where vclocks come in. The vector clocks used in Riak are based on the [work of Leslie Lamport](#).

When a value is stored in Riak, it is tagged with a vector clock, establishing its initial version. For each update, the vector clock is extended in such a way that Riak can later compare to versions of the object and determine:

1. Whether one object is a direct descendant of the other.

2. Whether the objects are direct descendants of a common parent.
3. Whether the objects are unrelated in recent heritage.

Using this knowledge, Riak can possibly auto-repair out-of-sync data, or at least provide a client with an opportunity to reconcile divergent changesets in an application specific manner.

Riak attempts to move data toward a consistent state across nodes, but it doesn't do so by comparing each and every object on each node. Instead, nodes gossip a [Merkle tree](#), which allows them to quickly decide which values need comparing.

Backends

Sharing data among nodes, on rings, etc. is all well and good, but at some point, it has to actually be stored somewhere - like on disk! Because Riak is relevant to a wide variety of applications, its "backend" storage system is a pluggable one.

Each node may be configured with a different Erlang module for doing the simple storage, at the vnode level, below all of the interconnected cluster details. At the backend level, a module only needs to define "get", "put", "delete", and "keys list" functions that receive a key and value which are both Erlang binaries. The backend can consider these binaries completely opaque data, or examine them to make decisions about how best to store them.

The following backends are packaged with Riak:

1. `riak_kv_bitcask_backend` - stores data to bitcask
2. `riak_kv_fs_backend` - stores data directly to files in a nested directory structure on disk
3. `riak_kv_ets_backend` - stores data in ETS tables (which makes it volatile storage, but great for debugging)
4. `riak_kv_dets_backend` - stores data on-disk in DETS tables
5. `riak_kv_gb_trees_backend` - stores data using Erlang `gb_trees`
6. `riak_kv_cache_backend` - turns a bucket into a memcached-type memory cache, and ejects the least recently used objects either when the cache becomes full or the object's lease expires
7. `riak_kv_multi_backend` - configure per-bucket backends

More reading

More detailed information about aspects of how Riak works can be found on these pages:

- [Replication](#)
- [REST API](#)
- [Links](#)
- [MapReduce](#)
- [Configuration Files](#)
- [PBC API](#)
- [Erlang Client PBC](#)
- [Pre- and Post-Commit Hooks](#)

Replication

Replication is fundamental and automatic in Riak, providing security that your data will still be there if a node in your Riak cluster goes down. All data stored in Riak will be replicated to a number of nodes in the cluster according to the `n_val` property set on the bucket.

Selecting an N value (`n_val`)

By default, Riak chooses an `n_val` of 3 for you. This means that data stored in the bucket will be replicated to 3 different nodes. For this to be effective, you need at least 3 nodes in your cluster.

How to choose an N value depends largely on the application and the shape of the data. If your data is very transient and can be reconstructed easily by the application, choosing a lower N value will give you greater performance. If you need high assurance that data is available even after node failure, increasing the N value will help protect against loss. How many nodes do you expect will fail at any one time? Choose an N value larger than that and your data will still be accessible when they go down.

The N value also affects the behavior of read ([GET](#)) and write ([PUT](#)) requests. The tunable parameters you can submit with requests are bound by the N value. For example, if $N=3$, the maximum read quorum (R) you can request is also 3. If some nodes containing the data you are requesting are down, an R value larger than the number of available nodes with the data will cause the read to fail.

Setting the N value (`n_val`)

To change the N value for a bucket issue a PUT request to the bucket with the new value:

```
$ curl -X PUT -d '{"props":{"n_val":5}}' http://riak-host:8098/riak/bucket
```


Changing the N value after a bucket has data in it is *not recommended*. If you do change the value, especially increasing it, you might need to **force read repair**. Overwritten objects and newly stored objects will automatically be replicated to the correct number of nodes.

Read Repair

Read repair occurs when a successful read occurs – that is, the quorum was met – but not all replicas from which the object was requested agreed on the value. There are two possibilities here for the errant nodes:

1. The node responded with a "not found" for the object, meaning it doesn't have a copy.
2. The node responded with a **vector clock** that is an ancestor of the vector clock of the successful read.

When this situation occurs, Riak will force the errant nodes to update their object values based on the value of the successful read.

Forcing Read Repair

When you increase the `n_val` on the bucket, you may start to get failed read operations, especially if the `R` value you use is larger than the number of replicas that originally stored the object. Forcing read repair will solve this issue.

For each object that fails read (or the whole bucket, if you like), read the object using an `R` value less than or equal to the original number of replicas. For example, if your original `n_val` was 3 and you increased it to 5, perform your read operations with `R=3` or less. This will cause the nodes that do not have the object(s) yet to respond with "not found", invoking read repair.

So what does N=3 really mean?

`N=3` simply means that three copies of each piece of data will be stored in the cluster. That is, three different partitions/vnodes will receive copies of the data. **There are no guarantees that the three replicas will go to three separate physical nodes**; however, the built-in functions for determining where replicas go attempts to distribute the data evenly.

As nodes are added and removed from the cluster, the ownership of partitions changes and *may* result in an uneven distribution of the data. On some rare occasions, Riak will also aggressively reshuffle ownership of the partitions to achieve a more even balance.

For cases where the number of nodes is less than the `N` value, data will likely be duplicated on some nodes. For example, with `N=3` and 2 nodes in the cluster, one node will likely have one replica, and the other node will have two replicas.

REST API

This is an overview of the operations you can perform over the HTTP interface to Riak, and can be used as a guide for developing a compliant client. All URLs assume the default configuration values where applicable. All examples use curl.



Client ID

All requests should include the `X-Riak-ClientId` header, which can be any string that uniquely identifies the client, for purposes of tracing object modifications in the **vector clock**.



URL Escaping

Buckets, keys, and link specifications may not contain unescaped slashes. Use a URL-escaping library or replace slashes with `%2F`.

- Bucket operations
 - Read bucket information
 - Set bucket properties
- Object/key operations
 - Read object
 - Store a new object without a key
 - Store a new or existing object with a key
 - Delete object
- Query operations
 - Link-walking
 - MapReduce
- Other operations
 - Server status

Bucket operations

Read bucket information

Reads the bucket properties and/or keys.

Request

```
GET /riak/bucket
```

Optional query parameters:

- `props=[true|false]` - whether to return the bucket properties
- `keys=[true|false|stream]` - whether to return the keys stored in the bucket

Response

- Normal status codes: 200 OK
- Content-Type: `application/json`

The JSON object in the response will contain up to two entries, "props" and "keys", which are present or missing, according to the optional query parameters. The default is for both to be present.

If `keys=stream`, the response will be transferred using chunked-encoding, where each chunk is a JSON object. The first chunk will contain the "props" entry (if `props` was not set to false). Subsequent chunks will contain individual JSON objects with the "keys" entry containing a sublist of the total keyset (some sublists may be empty).

Example

```
$ curl -v http://127.0.0.1:8098/riak/test
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 200 OK
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Tue, 09 Mar 2010 20:52:09 GMT
< Content-Type: application/json
< Content-Length: 266
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"props":{"name":"test","allow_mult":false,"big_vclock":50,"chash_keyfun":{"mod":"riak_util",
"fun":"chash_std_keyfun"},"linkfun":{"mod":"raw_link_walker_resource","fun":"mapreduce_linkfun"},
"n_val":3,"old_vclock":86400,"small_vclock":10,"young_vclock":20},"keys":[]}
```

Set bucket properties

Sets bucket properties like `n_val` and `allow_mult`.

Request

```
PUT /riak/bucket
```

Important headers:

- Content-Type: `application/json`

The body of the request should be a JSON object with a single entry "props". Unmodified bucket properties may be omitted.

Useful properties:

- `n_val` - the number of replicas for objects in this bucket
- `allow_mult` - whether to allow sibling objects to be created (concurrent updates)

Other valid properties may be inferred from [reading the properties](#), but are less-frequently changed.

Response

- Normal status codes: 204 No Content
- Typical error codes
 - 400 Bad Request if the submitted JSON is invalid
 - 415 Unsupported Media Type if the Content-Type was not set to application/json in the request

If successful, no content will be returned in the response body.

Example

```
$ curl -v -X PUT -H "Content-Type: application/json" -d '{"props":{"n_val":5}}' http://127.0.0.1:8098/riak/test
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> PUT /riak/test HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
> Content-Type: application/json
> Content-Length: 21
>
< HTTP/1.1 204 No Content
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Tue, 09 Mar 2010 21:03:52 GMT
< Content-Type: application/json
< Content-Length: 0
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Object/key operations

Read object

Reads an object from a bucket.

Request

```
GET /riak/bucket/key
```

Important headers:

- `Accept` - When "multipart/mixed" is the preferred content-type, objects with siblings will return all siblings in single request. See [Get all siblings in one request](#) example. See also RFC 2616 - [Accept header definition](#).

Optional headers:

- `If-None-Match` and `If-Modified-Since` invoke conditional request semantics, matching on the `ETag` and `Last-Modified` of the object, respectively. If the object fails one of the tests (that is, if the `ETag` is equal or the object is unmodified since the supplied timestamp), Riak will return a `304 Not Modified` response. See also RFC 2616 - [304 Not Modified](#).

Optional query parameters:

- `r=2` - (read quorum) how many replicas need to agree when retrieving the object (default is 2)
- `vtag=...` - when accessing an object with siblings, which sibling to retrieve. See [Manually requesting siblings](#) example.

Response

Normal response codes: 200 OK, 300 Multiple Choices, 304 Not Modified

Typical error codes: 404 Not Found

Important headers:

- `Content-Type` - the media type/format

- `X-Riak-Vclock` - the opaque vector clock for the object
- `X-Riak-Meta-*` - any user-defined metadata defined when storing the object
- `ETag` - the entity tag for the object, useful for conditional GET operations and validation-based caching
- `Last-Modified` - a timestamp for when the object was last written, in HTTP datetime format
- `Link` - user- and system-defined links to other resources. [Read more about Links](#)

The body of the response will be the contents of the object except when [siblings](#) are present.



Siblings

When `allow_mult` is set to true in the [bucket properties](#), concurrent updates are allowed to create "sibling" objects, meaning that the object has any number of different values that are related to one another by the vector clock. This allows your application to use its own conflict resolution technique.

An object with multiple values will result in a 300 `Multiple Choices` response. If the `Accept` header prefers "multipart/mixed", all siblings will be returned in a single request as chunks of the "multipart/mixed" response body. Otherwise, a list of "vtags" will be given in a simple text format. You can request individual siblings by adding the `vtag` query parameter. See the [Siblings examples](#) for more information.

To resolve the conflict, [store](#) the resolved version with the `X-Riak-Vclock` given in the response.

Simple Example

```
$ curl -v http://127.0.0.1:8098/riak/test/doc2
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test/doc2 HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Riak-Vclock: a85hYGBgzGDKBVIsbLvmlWYwJTLmsTLcjeE5ypcFAA==
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Link: </riak/test>; rel="up"
< Last-Modified: Wed, 10 Mar 2010 18:11:41 GMT
< ETag: 6dQBm9oYAlmxRSH0e9615W
< Date: Wed, 10 Mar 2010 18:11:52 GMT
< Content-Type: application/json
< Content-Length: 13
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"foo":"bar"}
```

Siblings examples

Manually requesting siblings

```

$ curl -v http://127.0.0.1:8098/riak/test/doc
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test/doc HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 300 Multiple Choices
< X-Riak-Vclock:
a85hYGDgyGDKBVIszMk55zKYEhznWBlKIniO8kGF2TyvHYIKf0cIszUnMTBzHYVKbIhEUL+VK4spDFTPxhHzFyqhEoVQz7wkSAG
< Vary: Accept, Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Wed, 10 Mar 2010 18:12:29 GMT
< Content-Type: text/plain
< Content-Length: 102
<
Siblings:
16vic4eU9ny46o4KPiDz1f
4v5xOg4bVwUYZdMkqf0d6I
6nr5tDTmhxnwuAFJDD2s6G
6zRSZFUJLHXZ15o9CGOBYL
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0

$ curl -v http://127.0.0.1:8098/riak/test/doc?vtag=16vic4eU9ny46o4KPiDz1f
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test/doc?vtag=16vic4eU9ny46o4KPiDz1f HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Riak-Vclock:
a85hYGDgyGDKBVIszMk55zKYEhznWBlKIniO8kGF2TyvHYIKf0cIszUnMTBzHYVKbIhEUL+VK4spDFTPxhHzFyqhEoVQz7wkSAG
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Link: </riak/test>; rel="up"
< Last-Modified: Wed, 10 Mar 2010 18:01:06 GMT
< ETag: 16vic4eU9ny46o4KPiDz1f
< Date: Wed, 10 Mar 2010 18:14:11 GMT
< Content-Type: application/x-www-form-urlencoded
< Content-Length: 13
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"bar":"baz"}

```

Get all siblings in one request

```

$ curl -v http://127.0.0.1:8098/riak/test/doc -H "Accept: multipart/mixed"
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test/doc HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: multipart/mixed
>
< HTTP/1.1 300 Multiple Choices
< X-Riak-Vclock:
a85hYGDgyGDKBVIsmk55zKYEhnzWBlKIniO8kGF2TyvHYIKf0cIszUnMTBzHYVKbIhEU1+VK4spDFTPxhHzFyqhEoVQz7wkSAG
< Vary: Accept, Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Wed, 10 Mar 2010 18:13:06 GMT
< Content-Type: multipart/mixed; boundary=YinLMzyUR9feB17okMytgKsylvh
< Content-Length: 766
<

--YinLMzyUR9feB17okMytgKsylvh
Content-Type: application/x-www-form-urlencoded
Link: </riak/test>; rel="up"
Etag: 16vic4eU9ny46o4KPiDz1f
Last-Modified: Wed, 10 Mar 2010 18:01:06 GMT

{"bar":"baz"}
--YinLMzyUR9feB17okMytgKsylvh
Content-Type: application/json
Link: </riak/test>; rel="up"
Etag: 4v5xOg4bVwUYZdMkqf0d6I
Last-Modified: Wed, 10 Mar 2010 18:00:04 GMT

{"bar":"baz"}
--YinLMzyUR9feB17okMytgKsylvh
Content-Type: application/json
Link: </riak/test>; rel="up"
Etag: 6nr5tDTmhxnwuAFJDD2s6G
Last-Modified: Wed, 10 Mar 2010 17:58:08 GMT

{"bar":"baz"}
--YinLMzyUR9feB17okMytgKsylvh
Content-Type: application/json
Link: </riak/test>; rel="up"
Etag: 6zRSZFUJlHXZ15o9CG0BYl
Last-Modified: Wed, 10 Mar 2010 17:55:03 GMT

{"foo":"bar"}
--YinLMzyUR9feB17okMytgKsylvh--
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0

```

Store a new object without a key

Stores a new object in a bucket with a random Riak-assigned key.

Request

```
POST /riak/bucket
```

Important headers:

- Content-Type must be set for the stored object. Set what you expect to receive back when next requesting it.
- X-Riak-Meta-* any additional metadata headers that should be stored with the object.
- Link - user- and system-defined links to other resources. [Read more about Links](#)

Optional query parameters:

- `w=2` (write quorum) how many replicas to write to before returning a successful response (default is 2).
- `dw=0` how many replicas to commit to durable storage before returning a successful response (default is 0)
- `returnbody=[true|false]` whether to return the contents of the stored object.

This request must include a body (entity).

Response

Normal status codes: 201 Created

Important headers:

- `Location` a relative URL to the newly-created object

If `returnbody=true`, any of the response headers expected from [Read object](#) may be present.

Example

```
$ curl -v -d 'this is a test' -H "Content-Type: text/plain" http://127.0.0.1:8098/riak/test
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> POST /riak/test HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
> Content-Type: text/plain
> Content-Length: 14
>
< HTTP/1.1 201 Created
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Location: /riak/test/bzPygTesROptGGVUKfyvp2RR49
< Date: Wed, 10 Mar 2010 13:12:54 GMT
< Content-Type: application/json
< Content-Length: 0
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Store a new or existing object with a key

Stores an object in a bucket with an existing or user-defined key.

Request

```
PUT /riak/bucket/key
```

POST is also a valid verb, for compatibility's sake.

Important headers:

- `Content-Type` must be set for the stored object. Set what you expect to receive back when next requesting it.
- `X-Riak-Vclock` if the object already exists, the vector clock attached to the object when read
- `X-Riak-Meta-*` any additional metadata headers that should be stored with the object.
- `Link` - user- and system-defined links to other resources. [Read more about Links](#)

Optional headers:

- `If-None-Match`, `If-Match`, `If-Modified-Since`, and `If-Unmodified-Since` invoke conditional request semantics, matching on the `ETag` and `Last-Modified` of the existing object. These can be used to enforce uniqueness or to prevent overwriting a modified object. If the test fails, you will receive a 412 `Precondition Failed` response.

Optional query parameters:

- `r=2` (read quorum) how many replicas need to agree when retrieving an existing object before the write (default is 2)

- `w=2` (write quorum) how many replicas to write to before returning a successful response (default is 2).
- `dw=0` how many replicas to commit to durable storage before returning a successful response (default is 0)
- `returnbody=[true|false]` whether to return the contents of the stored object.

Response

Normal status codes: 200 OK, 204 No Content, 300 Multiple Choices

Typical error codes:

- 412 Precondition Failed if one of the conditional request headers failed to match (see above)

If `returnbody=true`, any of the response headers expected from [Read object](#) may be present. Like [Read object](#), 300 Multiple Choices may be returned if siblings existed or were created as part of the operation, and the response can be dealt with similarly.

Example

```
$ curl -v -X PUT -d '{"bar":"baz"}' -H "Content-Type: application/json" -H "X-Riak-Vclock:
a85hYGBgzGDKBVISzMk55zKYEhnzWBlKIniO8mUBAA==" http://127.0.0.1:8098/riak/test/doc?returnbody=true
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> PUT /riak/test/doc?returnbody=true HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
> Content-Type: application/json
> X-Riak-Vclock: a85hYGBgzGDKBVISzMk55zKYEhnzWBlKIniO8mUBAA==
> Content-Length: 13
>
< HTTP/1.1 200 OK
< X-Riak-Vclock: a85hYGBgzGDKBVISzMk55zKYEhnzWBlKIniO8kGF2TyvHYIKfwcJZwEA
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Link: </riak/test>; rel="up"
< Date: Wed, 10 Mar 2010 17:55:03 GMT
< Content-Type: application/json
< Content-Length: 13
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"bar":"baz"}
```

Delete object

Deletes an object from a bucket.

Request

```
DELETE /riak/bucket/key
```

Response

- Normal response codes: 204 No Content, 404 Not Found

404 responses are "normal" in the sense that DELETE operations are idempotent and not finding the resource has the same effect as deleting it.

Example


```
$ curl -v -X DELETE http://127.0.0.1:8098/riak/test/test2
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> DELETE /riak/test/test2 HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 204 No Content
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Wed, 10 Mar 2010 12:53:33 GMT
< Content-Type: application/json
< Content-Length: 0
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Query operations

Link-walking

Link walking finds and returns objects by following links attached to them, starting from the object specified by the bucket and key portion. It is a special case of [MapReduce](#), and can be expressed more verbosely as such. [Read more about Links](#).

Request

```
GET /riak/bucket/key/[link specs+]
```



Link specifications

A link specification within the request URL is made of three parts:

- Bucket - a bucket name to limit the links to
- Tag - a "riaktag" to limit the links to
- Keep - 0 or 1, whether to return results from this phase

Any of the three parts may be replaced with "_", signifying that any value is valid. Multiple phases of links can be followed by adding additional path segments to the URL, separating the link specifications by slashes. The final phase in the link-walking query implicitly returns its results.

Response

Normal status codes: 200 OK

Typical error codes: 404 Not Found - if the origin object of the walk was missing

Important headers:

- Content-Type - always "multipart/mixed", with a boundary specified



Understanding the response body

The response body will always be multipart/mixed, with each chunk representing a single phase of the link-walking query. Each phase will also be encoded in multipart/mixed, with each chunk representing a single object that was found. If no objects were found or "keep" was not set on the phase, no chunks will be present in that phase. Objects inside phase results will include `Location` headers that can be used to determine bucket and key. In fact, you can treat each object-chunk similarly to a complete response from [Read object](#), without the status code.

Example

```

$ curl -v http://127.0.0.1:8098/riak/test/doc3/test,_/1/_/next,1
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /riak/test/doc3/test,_/1/_/next,1 HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Expires: Wed, 10 Mar 2010 20:24:49 GMT
< Date: Wed, 10 Mar 2010 20:14:49 GMT
< Content-Type: multipart/mixed; boundary=JZi8W8pB0Z3nO3odw1lGUB4LQCN
< Content-Length: 970
<

--JZi8W8pB0Z3nO3odw1lGUB4LQCN
Content-Type: multipart/mixed; boundary=OjZ8Km9J5vbsmxtcnlp48J91cJP

--OjZ8Km9J5vbsmxtcnlp48J91cJP
X-Riak-Vclock:
a85hYGDgymDKBVISzmK55zKYEhnzWBlKIniO8kGF2TyvHYIKf0cIszUnMTBzHYVKbIhEU1+VK4spDFTPxhHzFyqhEoVQz7wkSAG
Location: /riak/test/doc
Content-Type: application/json
Link: </riak/test>; rel="up", </riak/test/doc2>; riaktag="next"
Etag: 3pvmY35coyWPxh8mh4uBQC
Last-Modified: Wed, 10 Mar 2010 20:14:13 GMT

{"riak":"CAP"}
--OjZ8Km9J5vbsmxtcnlp48J91cJP--

--JZi8W8pB0Z3nO3odw1lGUB4LQCN
Content-Type: multipart/mixed; boundary=RJKFlAs9PrdBNfd74HANycvbA8C

--RJKFlAs9PrdBNfd74HANycvbA8C
X-Riak-Vclock: a85hYGBgzGDKBVISbLvmlWYwJTLmsTlcjeE5ypcFAA==
Location: /riak/test/doc2
Content-Type: application/json
Link: </riak/test>; rel="up"
Etag: 6dQBm9oYAlmxRSH0e9615W
Last-Modified: Wed, 10 Mar 2010 18:11:41 GMT

{"foo":"bar"}
--RJKFlAs9PrdBNfd74HANycvbA8C--

--JZi8W8pB0Z3nO3odw1lGUB4LQCN--
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0

```

MapReduce

MapReduce is a generic way to query Riak by specifying inputs and constructing a set of map, reduce, and link phases through which data will flow. [Read more about MapReduce.](#)

Request

```
POST /mapred
```

Important headers:

- Content-Type - must always be "application/json". The format of the request body is described in detail on the [MapReduce](#) page.

Response

Normal status codes: 200 OK

Typical error codes: 400 Bad Request - if an invalid job is submitted.

Important headers:

- Content-Type - always "application/json"

Example

```
$ curl -v -d '{"inputs":"test",
"query":[{"link":{"bucket":"test"}},{map":{"language":"javascript","name":"Riak.mapValuesJson"}}]}'
-H "Content-Type: application/json" http://127.0.0.1:8098/mapred
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> POST /mapred HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: */*
> Content-Type: application/json
> Content-Length: 117
>
< HTTP/1.1 200 OK
< Server: MochiWeb/1.1 WebMachine/1.6 (eat around the stinger)
< Date: Wed, 10 Mar 2010 20:47:47 GMT
< Content-Type: application/json
< Content-Length: 30
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
[{"foo":"bar"}, {"riak":"CAP"}]
```

Other operations

Server status

Reports about the performance and configuration of the Riak node to which it was requested. You must have the `{riak_kv_stat, true}` configuration setting in `app.config` for this endpoint to be active.

Request

```
GET /stats
```

Important headers:

- Accept determines whether the response will be formatted in `application/json` or `text/plain`.

Response

- Normal status codes: 200 OK
- Content-Type: `application/json` or `text/plain` (JSON with added line-breaks)

Example

```
$ curl -v http://127.0.0.1:8098/stats -H "Accept: text/plain"
* About to connect() to 127.0.0.1 port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> GET /stats HTTP/1.1
> User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
> Host: 127.0.0.1:8098
> Accept: text/plain
>
< HTTP/1.1 200 OK
< Vary: Accept, Accept-Encoding
```

```

< Server: MochiWeb/1.1WebMachine/1.6 (eat around the stinger)
< Date: Tue, 09 Mar 2010 21:27:32 GMT
< Content-Type: text/plain
< Content-Length: 1548
<
{
  "vnode_gets": 0,
  "vnode_puts": 0,
  "vnode_gets_total": 10,
  "vnode_puts_total": 5,
  "node_gets": 0,
  "node_gets_total": 2,
  "node_get_fsm_time_mean": "undefined",
  "node_get_fsm_time_median": "undefined",
  "node_get_fsm_time_95": "undefined",
  "node_get_fsm_time_99": "undefined",
  "node_get_fsm_time_100": "undefined",
  "node_puts": 0,
  "node_puts_total": 1,
  "node_put_fsm_time_mean": "undefined",
  "node_put_fsm_time_median": "undefined",
  "node_put_fsm_time_95": "undefined",
  "node_put_fsm_time_99": "undefined",
  "node_put_fsm_time_100": "undefined",
  "cpu_nprocs": 64,
  "cpu_avg1": 82,
  "cpu_avg5": 84,
  "cpu_avg15": 72,
  "mem_total": 7326660000.0,
  "mem_allocated": 4873620000.0,
  "nodename": "riak@127.0.0.1",
  "connected_nodes": [

],
  "sys_driver_version": "1.5",
  "sys_global_heaps_size": 0,
  "sys_heap_type": "private",
  "sys_logical_processors": 2,
  "sys_otp_release": "R13B04",
  "sys_process_count": 160,
  "sys_smp_support": true,
  "sys_system_version": "Erlang R13B04 (erts-5.7.5) [source] [64-bit] [smp:2:2] [rq:2]
[async-threads:5] [hipe] [kernel-poll:true]\n",
  "sys_system_architecture": "i386-apple-darwin10.2.0",
  "sys_threads_enabled": true,
  "sys_thread_pool_size": 5,
  "sys_wordsize": 8,
  "ring_members": [
    "riak@127.0.0.1"
  ],
  "ring_num_partitions": 64,
  "ring_ownership": "[{'riak@127.0.0.1',64}]",
  "ring_creation_size": 64,
  "storage_backend": "innostore_riak"
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0

```

Links

Links are metadata that establish one-way relationships between objects in Riak.

The Link Header

The essential way to read and modify links via the [REST API](#) is the [HTTP Link header](#). This header emulates the purpose of `<link>` tags in HTML, that is, establishing relationships to other HTTP resources. The format that Riak uses is like so:

```
Link: </riak/bucket/key>; riaktag="tag"
```

Inside the angle-brackets (`<`, `>`) is a relative URL to another object in Riak. The "tag" portion in double-quotes is any string identifier that has a meaning relevant to your application.

Objects can have multiple links by separating them with commas. For example, if an object was a participant in a doubly-linked list of objects, it might look like this:

```
Link: </riak/list/1>; riaktag="previous", </riak/list/3>; riaktag="next"
```

NOTE: There is no artificial limit to the number of links an object can have. But, as adding links to an object does increase that object's size, the same guidelines that apply to your data should also apply to your links: strike a balance between size and usability.

Links in the Erlang API

Links in the Erlang API are stored as tuples in the object metadata of this form:

```
{<<"bucket">>, <<"key">>, <<"tag">>}
```

To access the links, use `riak_object:get_metadata/1` to retrieve the metadata dict, and then retrieve the `<<"Links">>` key from that dict. Example:

```
1> {ok, Object} = Client:get(<<"list">>, <<"2">>, 1).
2> Meta = riak_object:get_metadata(Object).
3> Links = dict:fetch(<<"Links">>, Meta).
[{{<<"list">>, <<"1">>}, <<"previous">>}, {{<<"list">>, <<"3">>}, <<"next">>}]
```

To store links back in the object, update the dict, update the object metadata, and put the object:

```
4> NewMeta = dict:store(<<"Links">>, [{{<<"list">>, <<"0">>}, <<"first">>} | Links], Meta).
5> NewObject = riak_object:update_metadata(Object, NewMeta).
6> Client:put(NewObject, 2).
```

Link-walking

Link-walking (traversal) is a special case of [MapReduce](#) querying, and can be accessed through the [REST API](#). Link-walks start at a single input object and follow links on that object to find other objects that match the submitted specifications. More than one traversal may be specified in a single request, with any number of the intermediate results returned. The final traversal in a link-walking request always returns results.

- See also [Blog post "Link-walking by Example"](#)

MapReduce

Riak's primary querying and data-processing system is an implementation of the MapReduce programming paradigm popularized by [Google](#).

- [Why MapReduce?](#)
- [An Introduction to MapReduce](#)
 - [How Riak Spreads Processing](#)
 - [How Riak's MapReduce Queries Are Specified](#)
 - [How a Map Phase Works in Riak](#)
 - [How a Reduce Phase Works in Riak](#)
 - [How a Link Phase Works in Riak](#)
- [MapReduce via the REST API](#)
 - [REST Example](#)
 - [REST Query Syntax](#)
- [MapReduce via the Erlang API](#)
 - [Erlang Example](#)
 - [Erlang Query Syntax](#)
- [Phase functions](#)
 - [Map phase functions](#)
 - [Reduce phase functions](#)

Why MapReduce?

Key-value stores like Riak generally have very little functionality beyond just storing and fetching objects. MapReduce adds the capability to

perform more powerful queries over the data stored in Riak. It also fits nicely with the functional programming orientation of Riak's core code and the distributed nature of the data storage.

An Introduction to MapReduce

The main goal of MapReduce is to spread the processing of a query across many systems to take advantage of parallel processing power. This is generally done by dividing the query into several steps, dividing the dataset into several chunks, and then running those step/chunk pairs in separate physical hosts.

One step type is called "map". Map functions take one piece of data as input, and produce zero or more results as output. If you're familiar with "mapping over a list" in functional programming style, you're already familiar with "map" steps in a map/reduce query.

Another step type is called "reduce". The purpose of a "reduce" step is to combine the output of many "map" step evaluations, into one result.

The common example of a map/reduce query involves a "map" step that takes a body of text as input, and produces a word count for that body of text. A reduce step then takes the word counts produced from many bodies of text and either sums them to provide a word count for the corpus, or filters them to produce a list of documents containing only certain counts.

How Riak Spreads Processing

Riak's MapReduce has an additional goal: increasing data-locality. When processing a large dataset, it's often much more efficient to take the computation to the data than it is to bring the data to the computation.

It is Riak's solution to the data-locality problem that determines how Riak spreads the processing across the cluster. In the same way that any Riak node can coordinate a read or write by sending requests directly to the other nodes responsible for maintaining that data, any Riak node can also coordinate a MapReduce query by sending a map-step evaluation request directly to the node responsible for maintaining the input data. Map-step results are sent back to the coordinating node, where reduce-step processing can produce a unified result.

Put more simply: Riak runs map-step functions right on the node holding the input data for those functions, and it runs reduce-step functions on the node coordinating the MapReduce query.

How Riak's MapReduce Queries Are Specified

MapReduce queries in Riak have two components: a list of inputs and a list of "steps", or "phases".

Each element of the input list is a bucket-key pair. This bucket-key pair may also be annotated with "key-data", which will be passed as an argument to a map function, when evaluated on the object stored under that bucket-key pair.

Each element of the phases list is a description of a map function, a reduce function, or a link function. The description includes where to find the code for the phase function (for map and reduce phases), static data passed to the function every time it is executed during that phase, and a flag indicating whether or not to include the results of that phase in the final output of the query.

The phase list describes the chain of operations each input will flow through. That is, the initial inputs will be fed to the first phase in the list, and the output of that phase will be fed as input to the next phase in the list. This stream will continue through the final phase.

How a Map Phase Works in Riak

The input list to a map phase must be a list of (possibly annotated) bucket-key pairs. For each pair, Riak will send the request to evaluate the map function to the partition that is responsible for storing the data for that bucket-key. The vnode hosting that partition will lookup the object stored under that bucket-key, and evaluate the map function with the object as an argument. The other arguments to the function will be the annotation, if any is included, with the bucket-key, and the static data for the phase, as specified in the query.

How a Reduce Phase Works in Riak

Reduce phases accept any list of data as input, and produce any list of data as output. They also receive a phase-static value, specified in the query definition.

The important thing to understand is that the function defining the reduce phase may be evaluated multiple times, and the input of later evaluations will include the input of earlier evaluations.

For example, a reduce phase may implement the "set-union" function. In that case, the first set of inputs might be [1, 2, 2, 3], and the output would be [1, 2, 3]. When the phase receives more inputs, say [3, 4, 5], the function will be called with the concatenation of the two lists: [1, 2, 3, 3, 4, 5].

Other systems refer to the second application of the reduce function as a "re-reduce". There are at least a couple of reduce-query implementation strategies that work with Riak's model.

One strategy is to implement the phase preceding the reduce phase, such that its output is "the same shape" as the output of the reduce phase. This is how the examples in this document are written, and the way that we have found produces cleaner code.

An alternate strategy is to make the output of a reduce phase recognizable, such that it can be extracted from the input list on subsequent applications. For example, if inputs from the preceding phase are numbers, outputs from the reduce phase could be objects or strings. This

would allow the function to find the previous result, and apply new inputs to it.

How a Link Phase Works in Riak

Link phases find links matching patterns specified in the query definition. The patterns specify which buckets and tags links must have.

"Following a link" means adding it to the output list of this phase. The output of this phase is often most useful as input to a map phase, or another reduce phase.

MapReduce via the REST API

Riak supports writing MapReduce query functions in JavaScript and Erlang, as well as specifying query execution over HTTP.



"bad encoding" error

If you receive an error "bad encoding" from a MapReduce query that includes phases in Javascript, verify that your data does not contain incorrect Unicode escape sequences. Data being transferred into the Javascript VM must be in Unicode format.

REST Example

This example will store several chunks of text in Riak, and then compute word counts on the set of documents, using MapReduce via the REST API.

Load data

We will use the Riak HTTP interface to store the texts we want to process:

```
$ curl -X PUT -H "content-type: text/plain" \
  http://localhost:8098/riak/alice/p1 --data-binary @-
Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
it, 'and what is the use of a book,' thought Alice 'without pictures or
conversation?'
^D
$ curl -X PUT -H "content-type: text/plain" \
  http://localhost:8098/riak/alice/p2 --data-binary @-
So she was considering in her own mind (as well as she could, for the
hot day made her feel very sleepy and stupid), whether the pleasure
of making a daisy-chain would be worth the trouble of getting up and
picking the daisies, when suddenly a White Rabbit with pink eyes ran
close by her.
^D
$ curl -X PUT -H "content-type: text/plain" \
  http://localhost:8098/riak/alice/p5 --data-binary @-
The rabbit-hole went straight on like a tunnel for some way, and then
dipped suddenly down, so suddenly that Alice had not a moment to think
about stopping herself before she found herself falling down a very deep
well.
```

Run query

With data loaded, we can now run a query:

```
$ curl -X POST -H "content-type: application/json" http://localhost:8098/mapred --data @-
{"inputs": [["alice", "p1"], ["alice", "p2"], ["alice", "p5"]], "query": [{"map": {"language":
"javascript", "source": "function(v) { var m = v.values[0].data.toLowerCase().match('\\w*', 'g');
var r = []; for(var i in m) if (m[i] != '') { var o = {}; o[m[i]] = 1; r.push(o); } return r; }"}, {"reduce": {"language": "javascript", "source": "function(v) { var r = {}; for (var i in v) {
for(var w in v[i]) { if (w in r) r[w] += v[i][w]; else r[w] = v[i][w]; } } return [r]; }"}]}]}
^D
```

And we end up with the word counts for the three documents.

```
[{"the":8,"rabbit":2,"hole":1,"went":1,"straight":1,"on":2,"like":1,"a":6,"tunnel":1,"for":2,"some":1,"way":1,"and":5,"then":1,"dipped":1,"suddenly":3,"down":2,"so":2,"that":1,"alice":3,"had":3,"not":1,"moment":1,"to":3,"think":1,"about":1,"stopping":1,"herself":2,"before":1,"she":4,"found":1,"falling":1,"very":3,"deep":1,"well":2,"was":3,"considering":1,"in":2,"her":5,"own":1,"mind":1,"as":2,"could":1,"hot":1,"day":1,"made":1,"feel":1,"sleepy":1,"stupid":1,"whether":1,"pleasure":1,"of":5,"making":1,"daisy":1,"chain":1,"would":1,"be":1,"worth":1,"trouble":1,"getting":1,"up":1,"picking":1,"daisies":1,"when":1,"white":1,"with":1,"pink":1,"eyes":1,"ran":1,"close":1,"by":2,"beginning":1,"get":1,"tired":1,"sitting":1,"sister":2,"bank":1,"having":1,"nothing":1,"do":1,"once":1,"or":3,"twice":1,"peeped":1,"into":1,"book":2,"reading":1,"but":1,"it":2,"no":1,"pictures":2,"conversations":1,"what":1,"is":1,"use":1,"thought":1,"without":1,"conversation":1}]
```

Explanation

For more details about what each bit of syntax means, and other syntax options, read the following sections. As a quick explanation of how this example map/reduce query worked, though:

1. The objects named p1, p2, and p5 from the =alice= bucket were given as inputs to the query.
2. The map function from the phase was run on each object. The function:

```
function(v) {
  var m = v.values[0].data.match('\w*', 'g');
  var r = [];
  for(var i in m)
    if (m[i] != '') {
      var o = {};
      o[m[i]] = 1;
      r.push(o);
    }
  return r;
}
```

creates a list of JSON objects, one for each word (non-unique) in the text. The object has as a key, the word, and as the value for that key, the integer 1.

3. The reduce function from the phase was run on the outputs of the map functions. The function:

```
function(v) {
  var r = {};
  for (var i in v) {
    for(var w in v[i]) {
      if (w in r)
        r[w] += v[i][w];
      else
        r[w] = v[i][w];
    }
  }
  return [r];
}
```

looks at each JSON object in the input list. It steps through each key in each object, and produces a new object. That new object has a key for each key in every other object, the value of that key being the sum of the values of that key in the other objects. It returns this new object in a list, because it may be run a second time on a list including that object and more inputs from the map phase.

4. The final output is a list with one element: a JSON object with a key for each word in all of the documents (unique), with the value of that key being the number of times the word appeared in the documents.

REST Query Syntax

Map/Reduce queries are issued over HTTP via a POST to the /mapred resource. The body should be application/json of the form {"inputs":[...inputs...],"query":[...query...]}

Map/Reduce queries have a default timeout of 60000 milliseconds (60 seconds). The default timeout can be overridden by supplying a different value, in milliseconds, in the JSON document {"inputs":[...inputs...],"query":[...query...],"timeout": 90000}

Inputs

The list of input objects is given as a list of 2-element lists of the form [Bucket,Key] or 3-element lists of the form [Bucket,Key,KeyData].

You may also pass just the name of a bucket (`{"inputs": "mybucket", ...}`), which is equivalent to passing all of the keys in that bucket as inputs (i.e. "a map/reduce across the whole bucket"). You should be aware that this triggers the somewhat expensive "list keys" operation, so you should use it sparingly.

Query

The query is given as a list of phases, each phase being of the form `{PhaseType: { ... spec ...}}`. Valid `PhaseType` values are "map", "reduce", and "link".

Every phase `spec` may include a `keep` field, which must have a boolean value: `true` means that the results of this phase should be included in the final result of the map/reduce, `false` means the results of this phase should be used only by the next phase. Omitting the `keep` field accepts its default value, which is `false` for all phases except the final phase (Riak assumes that you were most interested in the results of the last phase of your map/reduce query).

Map

Map phases must be told where to find the code for the function to execute, and what language that function is in.

Function source can be specified directly in the query by using the "source" spec field. Function source can also be loaded from a pre-stored riak object by providing "bucket" and "key" fields in the spec. Erlang map functions can be specified using the "module" and "function" fields in the spec.

For example:

```
{"map": {"language": "javascript", "source": "function(v) { return [v]; }", "keep": true}}
```

would run the Javascript function given in the spec, and include the results in the final output of the m/r query.

```
{"map": {"language": "javascript", "bucket": "myjs", "key": "mymap", "keep": false}}
```

would run the Javascript function declared in the content of the Riak object under `mymap` in the `myjs` bucket, and the results of the function would not be included in the final output of the m/r query.

```
{"map": {"language": "erlang", "module": "riak_mapreduce", "function": "map_object_value"}}
```

would run the Erlang function `riak_mapreduce:map_object_value/3`.

Map phases may also be passed static arguments by using the "arg" spec field.

Reduce

Reduce phases look exactly like map phases, but are labeled "reduce".

Link

Link phases accept `bucket` and `tag` fields that specify which links match the link query. The string "_" (underscore) in each field means "match all", while any other string means "match exactly this string". If either field is left out, it is considered to be set to "_" (match all).

For example:

```
{"link": {"bucket": "foo", "keep": false}}
```

would follow all links pointing to objects in the `foo` bucket, regardless of their tag.



Riak comes with some prebuilt JavaScript Functions. You can check them out at: http://hg.basho.com/riak/src/tip/apps/riak_kv/priv/mapred_builtins.js

MapReduce via the Erlang API

Riak also supports invoking MapReduce queries via the Erlang API.



Distributing Erlang MapReduce Code

Any modules and functions you use in your Erlang MapReduce calls must be available on all nodes in the cluster. You can add them in Erlang applications in the `rel/overlay/lib` directory, by specifying the `-pz` option in `vm.args`, or by adding the path to the `add_paths` setting in `app.config`.

Erlang Example

If you have a Riak client hanging around, you can execute MapReduce queries on it like this:

```
1> Count = fun(G, undefined, none) ->
    [dict:from_list([I, 1] || I <- riak_object:get_value(G))]
    end.
2> Merge = fun(Gcounts, none) ->
    [lists:foldl(fun(G, Acc) ->
        dict:merge(fun(_, X, Y) -> X+Y end,
            G, Acc)
        end,
        dict:new(),
        Gcounts)]
    end.
3> {ok, [R]} = Client:mapred([<<"groceries">>, <<"mine">>],
    [<<"groceries">>, <<"yours">>]],
    [{map, {qfun, Count}, none, false},
    {reduce, {qfun, Merge}, none, true}]).
4> L = dict:to_list(R).
```

If the "mine" and "yours" objects in the groceries bucket had values of ["bread", "cheese"], ["bread", "butter"], the sequence of commands above would result in L being bound to [{"bread", 2}, {"cheese", 1}, {"butter", 1}].

Erlang Query Syntax

`riak_client:mapred/2` takes two lists as arguments. The first list contains bucket-key pairs, inputs to the MapReduce query. The second list contains the phases of the query.

Inputs

The input objects are given as a list of tuples in the format `{Bucket, Key}` or `{{Bucket, Key}, KeyData}`. `Bucket` and `Key` should be binaries, and `KeyData` can be any Erlang term. The former form is equivalent to `{{Bucket, Key}, undefined}`.

Query

The query is given as a list of map, reduce and link phases. Map and reduce phases are each expressed as tuples in the following form:

```
{Type, FunTerm, Arg, Keep}
```

`Type` is an atom, either `map` or `reduce`. `Arg` is a static argument (any Erlang term) to pass to each execution of the phase. `Keep` is either `true` or `false` and determines whether results from the phase will be included in the final value of the query. Riak assumes the final phase will return results.

`FunTerm` is a reference to the function that the phase will execute and takes any of the following forms:

- `{modfun, Module, Function}` where `Module` and `Function` are atoms that name an Erlang function in a specific module.
- `{qfun, Fun}` where `Fun` is a callable fun term (closure or anonymous function).
- `{jsfun, Name}` where `Name` is a binary that, when evaluated in Javascript, points to a built-in Javascript function.
- `{jsanon, Source}` where `Source` is a binary that, when evaluated in Javascript is an anonymous function.
- `{jsanon, {Bucket, Key}}` where the object at `{Bucket, Key}` contains the source for an anonymous Javascript function.

Link phases are expressed in the following form:

```
{link, Bucket, Tag, Keep}
```

`Bucket` is either a binary name of a bucket to match, or the atom `'_'`, which matches any bucket. `Tag` is either a binary tag to match, or the atom `'_'`, which matches any tag. `Keep` has the same meaning as in map and reduce phases.



There is a small group of prebuilt Erlang MapReduce functions available with Riak. Check them out here: http://bitbucket.org/basho/riak/src/tip/apps/riak_kv/src/riak_kv_mapreduce.erl

Phase functions

MapReduce phase functions have the same properties, arguments and return values whether you write them in Javascript or Erlang.

Map phase functions

Map functions take three arguments (in Erlang, arity-3 is required). Those arguments are:

1. **Value** : the value found at a key. This will be a Riak object, which in Erlang is defined and manipulated by the `riak_object` module. In Javascript, a Riak object looks like this:

```
{
  "bucket":BucketAsString,
  "key":KeyAsString,
  "vclock":VclockAsString,
  "values":[
    {
      "metadata":{
        "X-Riak-VTag":VtagAsString,
        "X-riak-Last-Modified":LastModAsString,
        ...other metadata...
      },
      "data":ObjectData
    },
    ...other metadata/data values (siblings)...
  ]
}
```

2. **KeyData** : key data that was submitted with the inputs to the query or phase.
3. **Arg** : a static argument for the entire phase that was submitted with the query.

A map phase should produce a list of results. You will see errors if the output of your map function is not a list. Return the empty list if your map function chooses not to produce output. If your map phase is followed by another map phase, the output of the function must be compatible with the input to a map phase - a list of bucket-key pairs or bucket-key-keydata triples.

Map function examples

These map functions return the value (data) of the object being mapped:

Erlang
<pre>fun(Value, _KeyData, _Arg) -> [riak_object:get_value(Value)].</pre>
Javascript
<pre>function(value, keydata, arg){ return [value.values[0].data]; }</pre>

These map functions filter their inputs based on the arg and return bucket-key pairs for a subsequent map phase:

Erlang
<pre>fun(Value, _KeyData, Arg) -> Key = riak_object:key(Value), Bucket = riak_object:bucket(Value), case erlang:byte_length(Key) of L when L > Arg -> [{Bucket,Key}]; _ -> [] end end.</pre>
Javascript
<pre>function(value, keydata, arg){ if(value.key.length > arg) return [[value.bucket, value.key]]; else return []; }</pre>

Reduce phase functions

Reduce functions take two arguments. Those arguments are:

1. `ValueList`: the list of values produced by the preceding phase in the MapReduce query.
2. `Arg`: a static argument for the entire phase that was submitted with the query.

A reduce function should produce a list of values, but it must also be true that the function is commutative, associative, and idempotent. That is, if the input list `[a,b,c,d]` is valid for a given `F`, then all of the following must produce the same result:

```
F([a,b,c,d])
F([a,d] ++ F([c,b]))
F([F([a]),F([c]),F([b]),F([d])])
```

Reduce function examples

These reduce functions assume the values in the input are numbers and sum them:

Erlang

```
fun(ValueList, _Arg) ->
  [lists:foldl(fun erlang:'+' /2, 0, List)].
```

Javascript

```
function(valueList, arg){
  return [valueList.reduce(
    function(acc, value){
      return acc + value;
    }, 0)];
}
```

These reduce functions sort their inputs:

Erlang

```
fun(ValueList, _Arg) ->
  lists:sort(ValueList).
```

Javascript

```
function(valueList, arg){
  return valueList.sort();
}
```

Configuration Files

Riak has two configuration files located in `etc/` if you are using a source install and in `/etc/riak` if you used a binary install. The files are `app.config` and `vm.args`.

The `app.config` file is used to set various attributes for the node such as the backend the node will use to store data. The `vm.args` file is used to pass parameters to the Erlang node such as the name or cookie of the Erlang node.

- [app.config](#)
- [vm.args](#)
- [Rebar Overlays](#)

app.config

Riak and the Erlang applications it depends on are configured by settings in the `app.config` file in the `etc` directory of the Riak node. The format of the file is similar to Erlang's ".app" files:

```
[
  {riak_core, [
    {ring_state_dir, "data/ring"}
    %% More riak_core settings...
  ]},
  {riak_kv, [
    {storage_backend, riak_dets_backend},
    {riak_kv_dets_backend_root, "data/dets"}

    %% More riak_kv settings...
  ]}
  %% Other application configurations...
].
```



Configuration changes in 0.10

Many configuration settings changed names and sections in the 0.10 release. Please backup your `app.config` file when upgrading and then copy your previous customizations to the proper places in the new file. See [the transition notes](#) for more information.

riak_core settings

default_bucket_props

properties to give each bucket, by default

Properties in this list will override hardcoded defaults in `riak_bucket:defaults/0`. This setting is the best way to set things like:

- the default N-value for Riak objects (`n_val`)
- whether or not siblings are allowed (`allow_mult`)
- the function for extracting links from objects (`linkfun`)

ring_state_dir

The directory on-disk in which to store the ring state (default: `"data/ring"`)

Riak's ring state is stored on-disk by each node, such that each node may be restarted at any time (purposely, or via automatic failover) and know what its place in the cluster was before it terminated, without needing immediate access to the rest of the cluster.

ring_creation_size

The number of partitions to divide the hash space into (default: 64)

By default, each Riak node will own `ring_creation_size/(number of nodes in the cluster)` partitions. It is generally a good idea to specify a `ring_creation_size` a few times the number of nodes in your cluster (e.g. specify 64-256 partitions for a 4-node cluster). This gives you room to expand the number of nodes in the cluster, without worrying about underuse due to owning too few partitions.

web_ip

The ip address on which Riak's HTTP interface should listen (default: `"127.0.0.1"`)

Riak's HTTP interface will not be started if this setting is not defined.

web_port

The port on which Riak's HTTP interface should listen (default: 8098)

Riak's HTTP interface will not be started if this setting is not defined.

riak_kv settings

raw_name

The base of the path in the URL exposing Riak's HTTP interface (default: `"riak"`)

The default value will expose data at `/riak/Bucket/Key`. For example, changing this

setting to "bar" would expose the interface at /bar/Bucket/Key.

storage_backend

The module name of the storage backend that Riak should use (default: `riak_kv_dets_backend`)

The storage format Riak uses is configurable. Riak will refuse to start if no storage backend is specified.

Available backends, and their additional configuration options are:

- `riak_kv_dets_backend`: data is stored in DETS files
 - `riak_kv_dets_backend_root`: root directory where the DETS files are stored (default: "data/dets")
- `riak_kv_ets_backend`: data is stored in ETS tables (in-memory)
- `riak_kv_gb_trees_backend`: data is stored in general balanced trees (in-memory)
- `riak_kv_fs_backend`: data is stored in binary files on the filesystem
 - `riak_kv_fs_backendroot`: root directory where the files are stored (ex: "/var/lib/riak/data")
- `riak_kv_multi_backend`: enables storing data for different buckets in different backends
 - Specify the backend to use for a bucket with `riak_client:set_bucket(BucketName, [{backend, BackendName}])` in Erlang or {"props":{"backend":"BackendName"}} in the REST API.
 - `multi_backend_default`: default backend to use if none is specified for a bucket (one of the `BackendName` atoms specified in the `multi_backend` setting)
 - `multi_backend`: list of backends to provide
 - Format of each backend specification is `{BackendName, BackendModule, BackendConfig}`, where `BackendName` is any atom, `BackendModule` is the name of the Erlang module implementing the backend (the same values you would provide as `storage_backend` settings), and `BackendConfig` is a parameter that will be passed to the `start/2` function of the backend module.
- `riak_kv_cache_backend`: a backend that behaves as an LRU-with-timed-expiry cache
 - `riak_kv_cache_backend_memory`: maximum amount of memory to allocate, in megabytes (default: 100)
 - `riak_kv_cache_backend_ttl`: amount by which to extend an object's expiry lease on each access, in seconds (default: 600)
 - `riak_kv_cache_backend_max_ttl`: maximum allowed lease time (default: 3600)



Dynamically Changing ttl

There is currently no way to dynamically change the ttl per bucket. The current work around would be to define multiple `riak_cache_backends` under `riak_multi_backend` with different ttl values.

add_paths

a list of paths to add to the Erlang code path

This setting is especially useful for allowing Riak to use external modules during map/reduce queries.

riak_kv_stat

Enables the statistics-aggregator if set to `true` (default: `true`).

riak_handoff_port

TCP port number for the handoff listener (default: 8099)

handoff_concurrency

Number of vnode, per physical node, allowed to perform handoff at once (default: 4).

js_vm_count

How many Javascript virtual machines to start (default: 8).

js_source_dir

Where to load user-defined built in Javascript functions (default: `unset`)

vm.args

Parameters for the Erlang node on which Riak runs are set in the `vm.args` file in the `etc` directory of the embedded Erlang node. Most of these settings can be left at their defaults until you are ready to tune performance.

Two settings you may be interested in right away, though, are `-name` and `-setcookie`. These control the Erlang node names (possibly

host-specific), and Erlang inter-node communication access (cluster-specific), respectively.

The format of the file is fairly loose: all lines that do not begin with the # character are concatenated, and passed to the `erl` on the command line, as is.

More details about each of these settings can be found in the [Erlang](#) documentation for the `erl` Erlang virtual machine.

Erlang Runtime Configuration Options

-name

the name of the Erlang node (default: `riak@127.0.0.1`)

The default value, `riak@127.0.0.1` will work for running Riak locally, but for distributed (multi-node) use, the value after the @ should be changed to the IP address of the machine on which the node is running.

If you have properly-configured DNS, the short-form of this name can be used (for example: `riak`). The name of the node will then be `riak@Host.Domain`.

-setcookie

the cookie of the Erlang node (default: `riak`)

Erlang nodes grant or deny access based on the sharing of a previously-shared cookie. You should use the same cookie for every node in your Riak cluster, but it should be a not-easily-guessed string unique to your deployment, to prevent non-authorized access.

-heart

enable `heart` node monitoring (default: `/disabled/`)

Heart will restart nodes automatically, should they crash. However, heart is so good at restarting nodes that it can be difficult to prevent it from doing so. Enable heart once you are sure that you wish to have the node restarted automatically on failure.

+K

enable kernel polling (default: `true`)

+A

number of threads in the async thread pool (default: 5)

-env

set host environment variables for Erlang

-smp

Enables Erlang's SMP support. *This is necessary for the innostore backend to work, even on single-processor systems.* (default: `enable`)

Rebar Overlays

If you are going to be rebuilding Riak often, you will want to edit the `vm.args` and `app.config` files in the `rel/overlays/etc` directory. The copies of those files in the `release (embedded node)` directory will be overwritten by the files in the `overlays` directory when a `make rel` or `rebar generate` command is issued.

PBC API

This is an overview of the operations you can perform over the Protocol Buffers Client (PBC) interface to Riak, and can be used as a guide for developing a compliant client.

Protocol

Riak listens on a TCP port (8087 by default) for incoming connections. Once connected the client can send a stream of requests on the same connection.

Each operation consists of a request message and one or more response messages. Messages are all encoded the same way

- 32-bit length of message code + protocol buffer message in network order
- 8-bit message code to identify the protocol buffer message
- N-bytes of protocol buffers encoded message

Example

```

00 00 00 07 09 0A 01 62 12 01 6B
|----Len---|MC|----Message-----|

Len = 0x07
Message Code (MC) = 0x09 = RpbGetReq
RpbGetReq Message = 0x0A 0x01 0x62 0x12 0x01 0x6B

Decoded Message:
bucket: "b"
key: "k"

```

Message Codes

Message Code	Message Type
0	RpbErrorResp
1	RpbPingReq
2	RpbPingResp
3	RpbGetClientIdReq
4	RpbGetClientIdResp
5	RpbSetClientIdReq
6	RpbSetClientIdResp
7	RpbGetServerInfoReq
8	RpbGetServerInfoResp
9	RpbGetReq
10	RpbGetResp
11	RpbPutReq
12	RpbPutResp
13	RpbDelReq
14	RpbDelResp
15	RpbListBucketsReq
16	RpbListBucketsResp
17	RpbListKeysReq
18	RpbListKeysResp
19	RpbGetBucketReq
20	RpbGetBucketResp
21	RpbSetBucketReq
22	RpbSetBucketResp
23	RpbMapRedReq
24	RpbMapRedResp

Error Response

If the server experiences an error processing a request it will return an RpbErrorResp message instead expected RpbOpeationResp. The message contains an error string and an error code.

```
message RpbErrorResp {
  required bytes errormsg = 1;
  required uint32 errcode = 2;
}
```

Values

- errormsg- a string representation of what went wrong
- errcode- a numeric code. Currently only RIAKC_ERR_GENERAL=1 is defined.

Operations

Object/Key Operations

Read object

Read an object from Riak

Request

```
message RpbGetReq {

  required bytes bucket = 1;

  required bytes key = 2;

  optional uint32 r = 3;

}
```

Optional Parameters

- r- (read quorum) how many replicas need to agree when retrieving the object (default is 2)

Response

```
message RpbGetResp {

  repeated RpbContent content = 1;

  optional bytes vclock = 2;

}
```

- content- value+metadata entries for the object. If there are siblings there will be more than one entry. If the key is not found, content will be empty.
- vclock- vclock Opaque vector clock that must be included in RpbPutReq to resolve the siblings.

The content entries hold the object value and any metadata


```
// Content message included in get/put responses
/message RpbContent {
  required bytes value = 1;
  optional bytes content_type = 2;    // the media type/format
  optional bytes charset = 3;
  optional bytes content_encoding = 4;
  optional bytes vtag = 5;
  repeated RpbLink links = 6;        // links to other resources
  optional uint32 last_mod = 7;
  optional uint32 last_mod_usecs = 8;
  repeated RpbPair usermeta = 9;     // user metadata stored with the object
}
```

Each object can contain user-supplied metadata (X-Riak-Meta-* in the HTTP interface) consisting of a key/value pair. e.g. key=X-Riak-Meta-ACL value=users:r.administrators:f would allow an application to store access control information for it to enforce (**not** Riak).

```
// Key/value pair - used for user metadata
message RpbPair {
  required bytes key = 1;
  optional bytes value = 2;
}
```

Links store references to related bucket/keys and can be accessed through link walking in map/reduce.

```
// Link metadata
message RpbLink {
  optional bytes bucket = 1;
  optional bytes key = 2;
  optional bytes tag = 3;
}
```

 **Missing keys**
Remember - if a key is not stored in Riak an RpbGetResp without content and vclock fields will be returned. This should be mapped to whatever convention the client language uses to return not found, e.g. the erlang client returns an atom {error, notfound}

Example

Request

```
Hex      00 00 00 07 09 0A 01 62 12 01 6B
Erlang <<0,0,0,7,9,10,1,98,18,1,107>>

RpbGetReq protoc decode:
bucket: "b"
key: "k"
```

Response

```
Hex      00 00 00 4A 0A 0A 26 0A 02 76 32 2A 16 33 53 44
        6C 66 34 49 4E 4B 7A 38 68 4E 64 68 79 49 6D 4B
        49 72 75 38 BB D7 A2 DE 04 40 E0 B9 06 12 1F 6B
        CE 61 60 60 60 CC 60 CA 05 52 2C AC C2 5B 3F 65
        30 25 32 E5 B1 32 EC 56 B7 3D CA 97 05 00
Erlang <<0,0,0,74,10,10,38,10,2,118,50,42,22,51,83,68,108,102,52,73,78,75,122,
        56,104,78,100,104,121,73,109,75,73,114,117,56,187,215,162,222,4,64,
        224,185,6,18,31,107,206,97,96,96,96,204,96,202,5,82,44,172,194,91,63,
        101,48,37,50,229,177,50,236,86,183,61,202,151,5,0>>

RpbGetResp protoc decode:
content {
  value: "v2"
  vtag: "3SD1f4INKz8hNdhyImKIru"
  last_mod: 1271442363
  last_mod_usecs: 105696
}
vclock: "k\\316a`\\314`\\312\\005R,\\254\\302[?e0%2\\345\\2612\\354V\\267=\\312\\227\\005\\000"
```

Store a new or existing object with a key

Store a new or updated object under they bucket/key provided.

Request

```

message RpbPutReq {
    required bytes bucket = 1;
    required bytes key = 2;
    optional bytes vclock = 3;
    required RpbContent content = 4;
    optional uint32 w = 5;
    optional uint32 dw = 6;
    optional bool return_body = 7;
}

```

Required Parameters

- bucket- bucket key resides in
- key- key to create/update
- content- new/updated content for object - uses the same RpbContent message RpbGetResp returns data in and consists of metadata and a value.

Optional Parameters

- vclock- opaque vector clock provided by an earlier RpbGetResp message. Omit if this is a new key or you deliberately want to create a sibling.
- w- (write quorum) how many replicas to write to before returning a successful response (default is set by the server and is currently 2).
- dw- how many replicas to commit to durable storage before returning a successful response (default is set by the server and is currently 0)
- return_body- whether to return the contents of the stored object. Defaults to false.

Response

```

message RpbPutResp {
    repeated RpbContent contents = 1;
    optional bytes vclock = 2; // the opaque vector clock for the object
}

```

If returnbody is set to true on the put request, the RpbPutResp will contain the current object after the put completes.



N.B. this could contain siblings just like an RpbGetResp does.

Example

Request

```

Hex      00 00 00 1C 0B 0A 01 62 12 01 6B 22 0F 0A 0D 7B
         22 66 6F 6F 22 3A 22 62 61 72 22 7D 28 02 38 01
Erlang <<0,0,0,28,11,10,1,98,18,1,107,34,15,10,13,123,34,102,111,111,34,58,34,
         98,97,114,34,125,40,2,56,1>>

RpbPutReq protoc decode:
bucket: "b"
key: "k"
content {
  value: "{\\"foo\\"":\\"bar\\"}"
}
w: 2
return_body: true

```

Response

```

Hex      00 00 00 62 0C 0A 31 0A 0D 7B 22 66 6F 6F 22 3A
        22 62 61 72 22 7D 2A 16 31 63 61 79 6B 4F 44 39
        36 69 4E 41 68 6F 6D 79 65 56 6A 4F 59 43 38 AF
        B0 A3 DE 04 40 90 E7 18 12 2C 6B CE 61 60 60 60
        CA 60 CA 05 52 2C 2C E9 0C 86 19 4C 89 8C 79 AC
        0C 5A 21 B6 47 F9 20 C2 6C CD 49 AC 0D 77 7C A0
        12 FA 20 89 2C 00
Erlang <<0,0,0,98,12,10,49,10,13,123,34,102,111,111,34,58,34,98,97,114,34,125,
        42,22,49,99,97,121,107,79,68,57,54,105,78,65,104,111,109,121,101,86,
        106,79,89,67,56,175,176,163,222,4,64,144,231,24,18,44,107,206,97,96,
        96,96,202,96,202,5,82,44,44,233,12,134,25,76,137,140,121,172,12,90,33,
        182,71,249,32,194,108,205,73,172,13,119,124,160,18,250,32,137,44,0>>

RpbPutResp protoc decode:
contents {
  value: "{\"foo\\":\\"bar\\"}"
  vtag: "lcaykOD96iNAhomyeVjOYC"
  last_mod: 1271453743
  last_mod_usecs: 406416
}
vclock: "k\\316a`\\312`\\312\\005R, \\351\\014\\206\\031L\\211\\214y\\254\\014Z!\\266G\\371
\\3021\\315I\\254\\rw|\\240\\022\\372 \\211,\\000"

```

Delete object

Delete an object from a bucket

Request

```

message RpbDelReq {
  required bytes bucket = 1;
  required bytes key = 2;
  optional uint32 rw = 3;
}

```

Optional Parameters

- `rw` - how many replicas to delete before returning a successful response (default is set by the server and is currently 2).

Response

Only the message code is returned.

Example

Request

```

Hex      00 00 00 12 0D 0A 0A 6E 6F 74 61 62 75 63 6B 65
        74 12 01 6B 18 01
Erlang <<0,0,0,18,13,10,10,110,111,116,97,98,117,99,107,101,116,18,1,107,24,1>>

RpbDelReq protoc decode:
bucket: "notabucket"
key: "k"
rw: 1

```

Response

```

Hex      00 00 00 01 0E
Erlang <<0,0,0,1,14>>

RpbDelResp - only message code defined

```

Bucket Operations

List Buckets

List all of the bucket names available



Caution

This call can be expensive for the server - do not use in performance sensitive code

Request

Only the message code is required.

Response

```
message RpbListBucketsResp {
  repeated bytes buckets = 1;
}
```

Values

- buckets- buckets on the server

Example

Request

```
Hex      00 00 00 01 0F
Erlang <<0,0,0,1,15>>

RpbListBucketsReq - only message code defined
```

Response

```
Hex      00 00 00 2A 10 0A 02 62 31 0A 02 62 35 0A 02 62
          34 0A 02 62 38 0A 02 62 33 0A 03 62 31 30 0A 02
          62 39 0A 02 62 32 0A 02 62 36 0A 02 62 37
Erlang <<0,0,0,42,16,10,2,98,49,10,2,98,53,10,2,98,52,10,2,98,56,10,2,98,51,10,
          3,98,49,48,10,2,98,57,10,2,98,50,10,2,98,54,10,2,98,55>>

RpbListBucketsResp protoc decode:
buckets: "b1"
buckets: "b5"
buckets: "b4"
buckets: "b8"
buckets: "b3"
buckets: "b10"
buckets: "b9"
buckets: "b2"
buckets: "b6"
buckets: "b7"
```

List keys in bucket

List all of the keys in a bucket. This is a streaming call, with multiple response messages sent for each request.

Request

```
message RpbListKeysReq {
  required bytes bucket = 1;
}
```

Optional Parameters

- bucket- bucket to get keys from

Response

```
message RpbListKeysResp {
  repeated bytes keys = 1;
  optional bool done = 2;
}
```

Values

- keys- batch of keys in the bucket.
- done- set true on the last response packet

Example

Request

```
Hex      00 00 00 0B 11 0A 08 6C 69 73 74 6B 65 79 73
Erlang <<0,0,0,11,17,10,8,108,105,115,116,107,101,121,115>>

RpbListKeysReq protoc decode:
bucket: "listkeys"
```

Response Pkt 1

```
Hex      00 00 00 04 12 0A 01 34
Erlang <<0,0,0,4,18,10,1,52>>

RpbListKeysResp protoc decode:
keys: "4"
```

Response Pkt 2

```
Hex      00 00 00 08 12 0A 02 31 30 0A 01 33
Erlang <<0,0,0,8,18,10,2,49,48,10,1,51>>

RpbListKeysResp protoc decode:
keys: "10"
keys: "3"
```

Response Pkt 3

```
Hex      00 00 00 03 12 10 01
Erlang <<0,0,0,3,18,16,1>>

RpbListKeysResp protoc decode:
done: true
```

Get Bucket Properties

Get the properties for a bucket

Request

```
message RpbGetBucketReq {
  required bytes bucket = 1;
}
```

Required Parameters

- bucket- bucket to retrieve properties for

Response

```

message RpbGetBucketResp {
    required RpbBucketProps props = 1;
}
// Bucket properties
message RpbBucketProps {
    optional uint32 n_val = 1;
    optional bool allow_mult = 2;
}

```

Values

- n_val- current n_val for the bucket
- allow_mult- allow_mult set true if conflicts are returned to clients

Example

Request

```

Hex      00 00 00 0B 13 0A 08 6D 79 62 75 63 6B 65 74
Erlang <<0,0,0,11,19,10,8,109,121,98,117,99,107,101,116>>

RpbGetBucketReq protoc decode:
bucket: "mybucket"

```

Response

```

Hex      00 00 00 07 14 0A 04 08 05 10 01
Erlang <<0,0,0,7,20,10,4,8,5,16,1>>

RpbGetBucketResp protoc decode:
props {
  n_val: 5
  allow_mult: true
}

```

Set Bucket Properties

Set the properties for a bucket

Request

```

message RpbSetBucketReq {
    required bytes bucket = 1;
    required RpbBucketProps props = 2;
}
// Bucket properties
message RpbBucketProps {
    optional uint32 n_val = 1;
    optional bool allow_mult = 2;
}

```

Required Parameters

- bucket- bucket to set properties for
- props- updated properties - only set properties to change
- n_val- current n_val for the bucket
- allow_mult- allow_mult set true if conflicts are returned to clients

Response

Only the message code is returned.

Example

Change allow_mult to true for bucket "friends"

Request

```
Hex      00 00 00 0E 15 0A 07 66 72 69 65 6E 64 73 12 02
         10 01
Erlang <<0,0,0,14,21,10,7,102,114,105,101,110,100,115,18,2,16,1>>

RpbSetBucketReq protoc decode:
bucket: "friends"
props {
  allow_mult: true
}
```

Response

```
Hex      00 00 00 01 16
Erlang <<0,0,0,1,22>>

RpbSetBucketResp - only message code defined
```

Querying

Map/Reduce

Execute a map/reduce job.

Request

```
message RpbMapRedReq {
  required bytes request = 1;
  required bytes content_type = 2;
}
```

Required Parameters

- request- map/reduce job
- content_type- encoding for map/reduce job

Map/reduce jobs can be encoded in two different ways

- application/json- JSON-encoded map/reduce job
- application/x-erlang-binary- Erlang external term format

The JSON encoding is the same as [REST API](#) and the external term format is the same as the [local Erlang API](#)

Response

The results of the map/reduce job is returned for each phase that generates a result, encoded in the same format the job was submitted in. Multiple responses messages will be returned followed by a final message at the end of the job.

```
message RpbMapRedResp {
  optional uint32 phase = 1;
  optional bytes response = 2;
  optional bool done = 3;
}
```

Values

- phase- phase number of the map/reduce job
- response- response encoded with the content_type submitted
- done- set true on the last response packet

Example

Here is how submitting a JSON encoded job to sum up a bucket full of JSON encoded values.


```
{ "inputs": "bucket_501653",
  "query":
    [ { "map": { "arg": null,
                "name": "Riak.mapValuesJson",
                "language": "javascript",
                "keep": false } },
      { "reduce": { "arg": null,
                   "name": "Riak.reduceSum",
                   "language": "javascript",
                   "keep": true } } ] }
```

Request

```
Hex      00 00 00 F8 17 0A E2 01 7B 22 69 6E 70 75 74 73
         22 3A 20 22 62 75 63 6B 65 74 5F 35 30 31 36 35
         33 22 2C 20 22 71 75 65 72 79 22 3A 20 5B 7B 22
         6D 61 70 22 3A 20 7B 22 61 72 67 22 3A 20 6E 75
         6C 6C 2C 20 22 6E 61 6D 65 22 3A 20 22 52 69 61
         6B 2E 6D 61 70 56 61 6C 75 65 73 4A 73 6F 6E 22
         2C 20 22 6C 61 6E 67 75 61 67 65 22 3A 20 22 6A
         61 76 61 73 63 72 69 70 74 22 2C 20 22 6B 65 65
         70 22 3A 20 66 61 6C 73 65 7D 7D 2C 20 7B 22 72
         65 64 75 63 65 22 3A 20 7B 22 61 72 67 22 3A 20
         6E 75 6C 6C 2C 20 22 6E 61 6D 65 22 3A 20 22 52
         69 61 6B 2E 72 65 64 75 63 65 53 75 6D 22 2C 20
         22 6C 61 6E 67 75 61 67 65 22 3A 20 22 6A 61 76
         61 73 63 72 69 70 74 22 2C 20 22 6B 65 65 70 22
         3A 20 74 72 75 65 7D 7D 5D 7D 12 10 61 70 70 6C
         69 63 61 74 69 6F 6E 2F 6A 73 6F 6E

Erlang <<0,0,0,248,23,10,226,1,123,34,105,110,112,117,116,115,34,58,32,34,98,
117,99,107,101,116,95,53,48,49,54,53,51,34,44,32,34,113,117,101,114,
121,34,58,32,91,123,34,109,97,112,34,58,32,123,34,97,114,103,34,58,32,
110,117,108,108,44,32,34,110,97,109,101,34,58,32,34,82,105,97,107,46,
109,97,112,86,97,108,117,101,115,74,115,111,110,34,44,32,34,108,97,
110,103,117,97,103,101,34,58,32,34,106,97,118,97,115,99,114,105,112,
116,34,44,32,34,107,101,101,112,34,58,32,102,97,108,115,101,125,125,
44,32,123,34,114,101,100,117,99,101,34,58,32,123,34,97,114,103,34,58,
32,110,117,108,108,44,32,34,110,97,109,101,34,58,32,34,82,105,97,107,
46,114,101,100,117,99,101,83,117,109,34,44,32,34,108,97,110,103,117,
97,103,101,34,58,32,34,106,97,118,97,115,99,114,105,112,116,34,44,32,
34,107,101,101,112,34,58,32,116,114,117,101,125,125,93,125,18,16,97,
112,112,108,105,99,97,116,105,111,110,47,106,115,111,110>>

RpbMapRedReq protoc decode:
request: "{ \"inputs\": \"bucket_501653\", \"query\": [ { \"map\": { \"arg\": null, \
\"name\": \"Riak.mapValuesJson\", \"language\": \"javascript\", \"keep\": false } }, \
{ \"reduce\": { \"arg\": null, \"name\": \"Riak.reduceSum\", \"language\": \
\"javascript\", \"keep\": true } } ] }"
content_type: "application/json"
```

Response 1 - result from phase 1

```
Hex      00 00 00 08 18 08 01 12 03 5B 39 5D
Erlang <<0,0,0,8,24,8,1,18,3,91,57,93>>

RpbMapRedResp protoc decode:
phase: 1
response: "[9]"
```

Response 2 - end of map/reduce job

```
Hex      00 00 00 03 18 18 01
Erlang <<0,0,0,3,24,24,1>>

RpbMapRedResp protoc decode:
done: true
```

Other Operations

Ping

Check if the server is alive

Request

Just the RpbPingReq message code. No request message defined.

Response

Just the RpbPingResp message code. No response message defined.

Example

Request

```
Hex      00 00 00 01 01
Erlang <<0,0,0,1,1>>
```

Response

```
Hex      00 00 00 01 02
Erlang <<0,0,0,1,2>>
```

Get Client Id

Get the client id used for this connection. Client ids are used for conflict resolution and each unique actor in the system should be assigned one. A client id is assigned randomly when the socket is connected and can be changed using SetClientId below.

Request

Just the RpbGetClientIdReq message code. No request message defined.

Response

```
// Get ClientId Request - no message defined, just send RpbGetClientIdReq message code
message RpbGetClientIdResp {
    required bytes client_id = 1; // Client id in use for this connection
}
```

Example

Request

```
Hex      00 00 00 01 03
Erlang <<0,0,0,1,3>>
```

Response

```
Hex      00 00 00 07 04 0A 04 01 65 01 B5
Erlang <<0,0,0,7,4,10,4,1,101,1,181>>

RpbGetClientIdResp protoc decode:
client_id: "\\001e\\001\\265"
```

Set Client Id

Set the client id for this connection. A library may want to set the client id if it has a good way to uniquely identify actors or across reconnects. This will reduce vector clock bloat.

Request

```
message RpbSetClientIdReq {
    required bytes client_id = 1; // Client id to use for this connection
}
```

Response

Example

Request

```
Hex      00 00 00 07 05 0A 04 01 65 01 B6
Erlang <<0,0,0,7,5,10,4,1,101,1,182>>

RpbSetClientIdReq protoc decode:
client_id: "\\001e\\001\\266"
```

Response

```
Hex      00 00 00 01 06
Erlang <<0,0,0,1,6>>

RpbSetClientIdResp - only message code defined
```

Server Info

Request

Just the RpbGetServerInfoReq message code. No request message defined.

Response

```
message RpbGetServerInfoResp {
    optional bytes node = 1;
    optional bytes server_version = 2;
}
```

Example

Request

```
Hex      00 00 00 01 07
Erlang <<0,0,0,1,7>>

RpbGetServerInfoReq - only message code defined
```

Response

```
Hex      00 00 00 17 08 0A 0E 72 69 61 6B 40 31 32 37 2E
          30 2E 30 2E 31 12 04 30 2E 31 30
Erlang <<0,0,0,23,8,10,14,114,105,97,107,64,49,50,55,46,48,46,48,46,49,18,4,48,
          46,49,48>>

RpbGetServerInfoResp protoc decode:
node: "riak@127.0.0.1"
server_version: "0.10"
```

Erlang Client PBC

Riak Protocol Buffers Client Usage Introduction

This document assumes that you have already started your Riak cluster. For instructions on that prerequisite, refer to [Installation and Setup](#).

Overview

To talk to riak, all you need is an Erlang node with the riak-erlang-client library (riakc) in its code path.

```
$ erl -pa $PATH_TO_RIAKC/ebin
```

You'll know you've done this correctly if you can execute the following commands and get a path to a beam file, instead of the atom 'non_existing':

```
1> code:which(riakc_pb_socket).  
".../riak-erlang-client/ebin/riakc_pb_socket.beam"
```

Or you can install the library into your code path using rebar

```
$ cd riak-erlang-client  
$ ./rebar install
```

Connecting

Once you have your node running, pass your Riak server nodename to `riak:client_connect/1` to connect and get a client. This can be as simple as:

```
1> {ok, Pid} = riakc_pb_socket:start_link("127.0.0.1", 8087).  
{ok,<0.56.0>}
```

Verify connectivity with the server using `ping/1`.

```
2> riakc_pb_socket:ping(Pid).  
pong
```

Storing New Data

Each bit of data in Riak is stored in a "bucket" at a "key" that is unique to that bucket. The bucket is intended as an organizational aid, for example to help segregate data by type, but Riak doesn't care what values it stores, so choose whatever scheme suits you. Buckets, keys and values are all binaries.

Before storing your data, you must wrap it in a `riakc_obj`:

```
3> Object = riakc_obj:new(<<"groceries">>, <<"mine">>, <<"eggs & bacon">>).  
{riakc_obj,<<"groceries">>,<<"mine">>,undefined,undefined,  
  {dict,0,16,16,8,80,48,  
    {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],...],  
     {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],...]}},  
  <<"eggs & bacon">>}
```

The `Object` refers to a key `<<"mine">>` in a bucket named `<<"groceries">>` with the value `<<"eggs & bacon">>`. Using the client you opened earlier, store the object:

```
5> riakc_pb_socket:put(Pid, Object).  
ok
```

If the return value of the last command was anything but the atom 'ok', then the store failed. The return value may give you a clue as to why the store failed, but check the [Troubleshooting](#) section below if not.

The object is now stored in Riak. `put/2` uses default parameters for storing the object. There is also a `put/3` call that takes a proplist of options.

Option	Description
{w, W}	the minimum number of nodes that must respond with success for the write to be considered successful. The default is currently set on the server at 2

{dw, DW}	the minimum number of nodes that must respond with success after durably storing the object for the write to be considered successful The default is currently set on the server at 0. returnbody - immediately do a get after the put and return a riakc_obj.
----------	---

```
6> riakc_pb_socket:put(Pid, AnotherObject,
    [{w, 2}, {dw, 1}, returnbody]).
{ok, {riakc_obj, <<"my bucket">>, <<"my key">>,
    <<107,206,97,96,96,96,206,96,202,5,82,44,140,62,169,115,
    50,152,18,25,243,88,25,...>>,
    [{dict,2,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],...]}},
    <<"my binary data">>]},
    {dict,0,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],...]}},
    undefined}}
```

Would make sure at least two nodes responded successfully to the put and at least one node has durably stored the value and an updated object is returned.

See riak/doc/architecture.txt for more information about W and DW values.

Fetching Data

At some point you'll want that data back. Using the same bucket and key you used before:

```
7> {ok, O} = riakc_pb_socket:get(Pid, <<"groceries">>,
    <<"mine">>).
{ok, {riakc_obj, <<"groceries">>, <<"mine">>,
    <<107,206,97,96,96,96,204,96,202,5,82,44,12,143,167,115,
    103,48,37,50,230,177,50,...>>,
    [{dict,2,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],...]}},
    <<"eggs & bacon">>]},
    {dict,0,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],...]}},
    undefined}}
```

Like 'put', there is a 'get' functions that takes options, get/3.

Option	Description
{r, R}	the minimum number of nodes that must respond with success for the read to be considered successful

Modifying Data

Say you had the "grocery list" from the examples above, reminding you to get <<"eggs & bacon">>, and you want to add <<"milk">> to it. The easiest way is:

```

8> {ok, Oa} = riakc_pb_socket:get(Pid, <<"groceries">>, <<"mine">>).
...
9> Ob = riakc_obj:update_value(Oa, <<"milk, ", (riakc_obj:get_value(O))/binary>>).
11> {ok, Oc} = riakc_pb_socket:put(Pid, Ob, [return_body]).
{ok, {riakc_obj, <<"groceries">>, <<"mine">>,
      <<107,206,97,96,96,96,206,96,202,5,82,44,12,143,167,115,
      103,48,37,50,230,177,50,...>>,
      [{dict,2,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],[],...]}},
        <<"milk, eggs & bacon">>]],
      {dict,0,16,16,8,80,48,
        {[[],[],[],[],[],[],[],[],[],[],[],[],[],...},
        {[[],[],[],[],[],[],[],[],[],[],[],[],[],...]}},
      undefined}}

```

That is, fetch the object from Riak, modify its value with `riakc_obj:update_value/2`, then store the modified object back in Riak. You can get your updated object to convince yourself that your list is updated:

Deleting Data

Throwing away data is quick and simple: just use the `delete/3` function.

```

10> riakc_pb_socket:delete(Pid, <<"groceries">>, <<"mine">>).
ok

```

As with get and put, delete can also take options

Option	Description
{rw, RW}	the number of nodes to wait for responses from

Issuing a delete for an object that does not exist returns just returns ok.

Encoding

The initial release of the erlang protocol buffers client treats all values as binaries. The caller needs to make sure data is serialized and deserialized correctly. The content type stored along with the object may be used to store the encoding. For example

```

decode_term(Object) ->
  case riakc_obj:get_content_type(Object) of
    <<"application/x-erlang-term">> ->
      try
        {ok, binary_to_term(riakc_obj:get_value(Object))}
      catch
        _:Reason ->
          {error, Reason}
      end;
    Ctype ->
      {error, {unknown_ctype, Ctype}}
  end.

encode_term(Object, Term) ->
  riakc_obj:update_value(Object, term_to_binary(Term, [compressed]),
    <<"application/x-erlang-term">>).

```

Siblings

If a bucket is configured to allow conflicts (`allow_mult=true`) then the result object may contain more than one result. The number of values can be returned with

```
1> riakc_obj:value_count(Obj).
2
```

The values can be listed with

```
2> riakc_obj:get_values(Obj).
\[\<<"{"k1\":"v1\}">>,<<"{"k1\":"v1\}">>\]
```

And the content types as

```
3> riakc_obj:get_content_types(Obj).
[]
```

Siblings are resolved by calling `riakc_obj:update_value` with the winning value on an object returned by `get` or `put` with `return_body`.

Listing Keys

Most uses of key-value stores are structured in such a way that requests know which keys they want in a bucket. Sometimes, though, it's necessary to find out what keys are available (when debugging, for example). For that, there is `list_keys`:

```
riakc_pb_socket:list_keys(Pid, <<"groceries">>).
{ok,[<<"mine">>]}
```

Note that keylist updates are asynchronous to the object storage primitives, and may not be updated immediately after a `put` or `delete`. This function is primarily intended as a debugging aid.

`list_keys/2` is just a convenience function around the streaming version of the call `stream_list_keys(Pid, Bucket)`.

```
2> riakc_pb_socket:stream_list_keys(Pid, <<"groceries">>).
{ok,87009603}
3> receive Msg1 \-> Msg1 end.
{87009603,{keys,[]}}
4> receive Msg2 \-> Msg2 end.
{87009603,done}
```

See `riakc_pb_socket:wait_for_listkeys` for an example of receiving.

Bucket Properties

Bucket properties can be retrieved and modified using `get_bucket/2` and `set_bucket/3`. The bucket properties are represented as a proplist. Only a subset of the properties can be retrieved and set using the protocol buffers interface - currently only `n_val` and `allow_mult`.

Here's an example of getting/setting properties

```
3> riakc_pb_socket:get_bucket(Pid, "transcripts").
{ok,[{n_val,3},{allow_mult,false}]}
4> riakc_pb_socket:set_bucket(Pid, "transcripts", [{n_val, 5}]).
ok
5> riakc_pb_socket:get_bucket(Pid, "transcripts").
{ok,[{n_val,5},{allow_mult,false}]}
6> riakc_pb_socket:set_bucket(Pid, "transcripts", [{n_val, 7}, {allow_mult, true}]).
ok
7> riakc_pb_socket:get_bucket(Pid, "transcripts").
{ok,[{n_val,7},{allow_mult,true}]}
```

Map/Reduce

Troubleshooting

If `start/2` or `start_link/2` return `{error,econnrefused}` the client could not connect to the server - make sure the protocol buffers interface is enabled on the server and the address/port is correct.

Pre- and Post-Commit Hooks

Overview

Pre-commit hooks are invoked before a `riak_object` is persisted. Pre-commit hooks can:

1. allow the write to occur with an unmodified object
2. modify the object
3. Fail the update and prevent any modifications

Post-commit hooks are notified after the fact and should not modify `riak_object`. Updating `riak_objects` in post-commit hooks can cause nasty feedback loops which will wedge the hook into an infinite cycle unless the hook functions are carefully written to detect and short-circuit such cycles.

Pre- and post-commit hooks are defined on a per-bucket basis and are stored in the target bucket's properties. They are run once per successful response to the client.

Configuration

Configuring either pre- or post-commit hooks is very easy. Simply add a reference to your hook function to the list of functions stored in the correct bucket property. Pre-commit hooks are stored under the bucket property `precommit`. Post-commit hooks use the bucket property `postcommit`.

Pre-commit hooks can be implemented as named Javascript functions or as Erlang functions. The configuration for each is given below:

```
Javascript: {"name": "Foo.beforeWrite"}
Erlang: {"mod": "foo", "fun": "beforeWrite"}
```

Post-commit hooks can be implemented in Erlang only. The reason for this restriction is Javascript cannot call Erlang code and, thus, is prevented from doing anything useful. This restriction will be revisited when the state of Erlang/Javascript integration is improved. Post-commit hooks use the same function reference syntax as pre-commit hooks.

See the `map/reduce` documentation for steps to define your own pre-defined Javascript named functions.

Pre-Commit Hooks

API & Behavior

Pre-commit hook functions should take a single argument, the `riak_object` being modified. Remember that deletes are also considered "writes" so pre-commit hooks will be fired when a delete occurs. Hook functions will need to inspect the object for the `X-Riak-Deleted` metadata entry to determine when a delete is occurring.

Erlang pre-commit functions are allowed three possible return values:

1. A `riak_object` – This can either be the same object passed to the function or an updated version. This allows hooks to modify the object before they are written.
2. `fail` – The atom `fail` will cause Riak to fail the write and send a 403 Forbidden along with a generic error message about why the write was blocked.
3. `{fail, Reason}` – The tuple `{'fail', Reason}` will cause the same behavior as in #2 with the addition of `Reason` used as the error text.

Javascript pre-commit functions should also take a single argument, the JSON encoded version of the `riak_object` being modified. The JSON format is exactly the same as Riak's `map/reduce`. Javascript pre-commit functions are allowed three possible return values:

1. A JSON encoded Riak object – Aside from using JSON, this is exactly the same as #1 for Erlang functions. Riak will automatically convert it back to its native format before writing.
2. "fail" – The Javascript string "fail" will cause Riak to fail the write in exactly the same way as #2 for Erlang functions.
3. {"fail": Reason} – The JSON hash will have the same effect as #3 for Erlang functions. Reason must be a Javascript string.

Chaining

The default value of the bucket `precommit` property is an empty list. Adding one or more pre-commit hook functions, as documented above, to the list will cause Riak to start evaluating those hook functions when bucket entries are created, updated, or deleted. Riak stops evaluating pre-commit hooks when a hook function fails the commit.

Post-Commit Hooks

API & Behavior

Post-commit hooks are run after the write has completed successfully. Specifically, the hook function is called by `riak_kv_put_fsm` immediately

before the calling process is notified of the successful write. Hook functions must accept a single argument, the `riak_object` instance just written. The return value of the function is ignored. As with pre-commit hooks, deletes are considered writes so post-commit hook functions will need to inspect object metadata for the presence of `X-Riak-Deleted` to determine when a delete has occurred.

Chaining

The default value of the bucket `postcommit` property is an empty list. Adding one or more post-commit hook functions, as documented above, to the list will cause Riak to start evaluating those hook functions immediately after data has been created, updated, or deleted. Each post-commit hook function runs in a separate process so it's possible for several hook functions, triggered by the same update, to execute in parallel. **All** post-commit hook functions are executed for each create, update, or delete.

Client Libraries

The list of Basho supported client libraries is ever-growing. If you don't see what you are looking for here, you may find it in the list of [Community Contributed Projects](#).

Basho is currently developing and supporting drivers for:

- Erlang
- Javascript
- Java
- PHP
- Python
- Ruby

Erlang

The local Erlang client is a tightly-integrated part of Riak and the Riak REST interface uses the Erlang client internally. You can find more information about the Erlang-native driver in the [edoc API](#).

The primary client for code that is not inside Riak is the PBC client:

- Erlang Protocol Buffers Client Documentation
- <http://hg.basho.com/riak-erlang-client>

Javascript

The officially supported Javascript client uses [jQuery](#) for communicating with Riak via Ajax.

<http://hg.basho.com/riak-javascript-client/>



Sample App

yakriak - Riak-powered Ajax-polling chatroom

Java

The officially supported Java client was created by Jon Lee.

<http://hg.basho.com/riak-java-client/>

<http://bitbucket.org/jonjlee/riak-java-client/wiki/Home>

PHP

The officially supported PHP driver is available on Bitbucket:

<http://hg.basho.com/riak-php-client/>



Sample App

There is a sample PHP app that you can check out if you want to see the PHP library in action. You can get it [here](#).

Python

The officially supported Python driver is available on Bitbucket:

<http://hg.basho.com/riak-python-client>

You can also find documentation for the Python driver here:

http://riak.basho.com/python_client_api/riak.html

Ruby

[Ripple/riak-client](#) is the officially supported Ruby driver for Riak. It includes a driver for all major Riak client operations and an optional library for doing rich document-style object modeling.

- [Documentation](#)
- [Wiki](#)
- [Client Gem](#)
- [Bug Tracker](#)

If you are interested in using Riak with Ruby, you might find value in:

- [Why Riak Should Power Your Next Rails App](#)

Community-Developed Libraries and Projects

Community-Developed Libraries

The Riak Community is developing at a break-neck pace, and the number of community-contributed libraries and drivers is growing right along side it. Here is a list of projects that may suit your programming needs or curiosities. (If you know of something that needs to be added or are developing something that you wish to see added to this list, let us know in the [Comment](#) section below.)

Go (Go is a language recently released by Google. Read more about it [here](#).)

- [riak.go](#) - a Riak Client for Go

Griffon

- [Riak Plugin for Griffon](#)

Java

- [Riak-Java-PB-Client](#) - Java Client Library for Riak based on the Protocol Buffers API

JavaScript

- [Riak-js](#)
- [Node-Riak](#)
- [Nori](#) - Experimental Riak HTTP Library for Node.js modeled after Ripple

.NET

- [Hebo](#)
- [Data.RiakClient](#)

Perl

- [Net::Riak](#)
- [AnyEvent-Riak](#) adapter

Python

- [txriak](#) - a Twisted module for communicating with Riak via the HTTP interface

Ruby

- [Riak Model](#)
- [Riak DataObjects](#) adapter
- [Riak_Client](#)
- [Riak Sessions](#) for rack
- [Riaktor](#) - Ruby client and object mapper for Riak
- [DataMapper Adapter](#) for Riak
- [Riak PB Client](#) - Riak Protocol Buffer Client in Ruby

- [Devise-Ripple](#) - An ORM strategy to use Devise with Riak

Scala

- [Riakka](#)
- [Ryu](#) - A `Tornado Whirlwind Kick` Scala Client

Other

- [Chimera](#) - An object mapper for Riak and Redis
- [Riak_Redis Backend](#)
- [Riak Homebrew Formula](#)
- [Riak Admin](#) - A Futon-like web interface for Riak
- [Briak](#) - A Sinatra-based web front-end browser for Riak
- [Using Nginx as a front-end for Riak](#)
- [Riak-fuse](#) - A FUSE Driver for Riak
- [RiakAWS](#) - A simple way to deploy a Riak cluster in the Amazon Cloud
- [riakfuse](#) - A distributed filesystem that uses riak as its backend store
- [riak-admin](#) - A Java program with GUI to browse and update a Riak database.
- [ebot](#) - A scalable Web Crawler that supports Riak as a backend.
- [riak-jscouch](#) - JSCouch examples done with Riak.

Riak Comparisons

The NoSQL space, as well as the database space in general, is growing ever-crowded. Because of this, we often find ourselves answering very high level questions from developers, prospects, and customers along the lines of, "How does Riak compare to this database?" or "What is the main difference between your replication strategy and this NoSQL Database?" So, we thought it would be a worthwhile exercise to make available very brief and objective comparisons to as many databases as possible. (The list below will be growing as soon as we have the time to grow it.)

Disclaimer: We tried to get this right, but software is complicated, and it changes rapidly. If you think we have made an error, please kindly correct us, and we will be happy to make the change.

- [Riak Compared to Cassandra](#)
- [Riak Compared to CouchDB](#)
- [Riak Compared to MongoDB](#)
- [Riak Compared to Neo4j](#)

Riak Compared to Cassandra

This is intended to be a brief, objective and technical comparison of Riak and Cassandra

- [High Level Differences](#)
- [Scaling Out](#)
- [Queries and Distributed Operations](#)
- [Detecting Conflicting Writes](#)
- [Flexible Data Model](#)
- [HTTP Based API](#)
- [Configurable Storage](#)

High Level Differences

Both Riak and Cassandra are based on Amazon's description of Dynamo (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html), but they take two very different approaches. Riak attempts to be a faithful implementation of Dynamo, including all key components mentioned in the paper, with the addition of some added functionality like links and Map/Reduce operations. Cassandra departs from the Dynamo paper by omitting vector clocks and moving from partition-based consistent hashing to key ranges, while adding functionality like order-preserving partitioners and range queries.

This leads to several major differences detailed below.

Scaling Out

Riak allows you to elastically grow and shrink your cluster while evenly balancing the load on each machine. Adding a new node to a Riak cluster takes one command 'bin/riak join'. When you add a new node, it immediately begins taking an equal share of the existing data from the other machines in the cluster, as well as an equal share of all new requests and data. This works because the available data space for your cluster is divided into partitions (64 by default). When you add a new node, it claims an equal share of the partitions, meaning that every other node has a few less partitions to worry about. When you remove a node, the reverse happens, with the removed node handing partitions back to the remaining nodes in the cluster evenly.

[Adding Nodes to Riak](#)

In contrast, nodes in a Cassandra cluster claim a range of data. When you add a machine to a Cassandra cluster, by default Cassandra will take on half the key range of whichever node has the largest amount of data stored. Alternatively, you can override this by specifying an InitialToken setting, providing more control over which range is claimed. In this case data is moved from the two nodes on the ring adjacent to the new node. As a result, if all nodes in an N-node cluster were overloaded, you would need to add N/2 new nodes. Cassandra also provides a tool ('nodetool loadbalance') that can be run in a rolling manner on each node to rebalance the cluster.

<http://wiki.apache.org/cassandra/Operations>

Queries and Distributed Operations

Both Riak and Cassandra allow you to access your data in a simple key/value model.

Riak also allows you to query your data using links and Javascript-based Map/Reduce:

- With links, you create lightweight pointers between your data, for example, from 'projects' to 'milestones' to 'tasks', and then select data along that hierarchy using simple client API commands.
- With Javascript-based Map/Reduce you can define a Map/Reduce operation with multiple map and reduce phases, and then pass the operation to a Riak cluster along with a set of starting keys. Riak works like a mini-Hadoop, running the Map/Reduce operation in real time and returning the results. Any data processing happens in parallel across all machines, and any operations that run on a specific piece of data run on the machine that holds that data.

<https://wiki.basho.com/display/RIAK/MapReduce>

Cassandra on the other hand, allows you to query your data through data ranges or access your data from Hadoop:

- To use data ranges, the Cassandra database must be configured to use the order-preserving partitioner. (This is a cluster-wide setting.) The order-preserving partitioner requires special care to configure data ranges via the InitialToken setting, otherwise hotspots could result.
- With the upcoming release of Cassandra 0.6, Hadoop will be able to use Cassandra directly from a Map/Reduce job.

<http://spyced.blogspot.com/2009/05/consistent-hashing-vs-order-preserving.html>

<https://svn.apache.org/repos/asf/cassandra/trunk/NEWS.txt>

Detecting Conflicting Writes

Riak tags each object with a vector clock, which can be used to detect when two processes try to update the same data at the same time, or to ensure that the correct data is stored after a network split.

<http://riak.basho.com/edoc/vclock.html>

In contrast, Cassandra tags data with a timestamp, and compares timestamps to determine which data is newer. If a client's timestamp is out of sync with the rest of the cluster, or if a client waits too long between reading and writing data, then it is possible to lose the data that was written in between.

<http://wiki.apache.org/cassandra/DataModel>

Flexible Data Model

Riak buckets are created on the fly when they are first accessed. This allows an application to evolve its data model easily.

[Data Storage in Riak](#)

In contrast, the Cassandra Keyspaces and Column Families (akin to Databases and Tables) are defined in an XML file. Changing the data-model at this level requires a rolling reboot of the entire cluster.

<http://wiki.apache.org/cassandra/DataModel>

HTTP Based API

Riak uses a REST-based API. Apart from making it easy to develop new Client libraries, this allows you to put an HTTP Cache or Proxy (like nginx) in front of a cluster for load balancing, caching, or access control. For example, your application could store images in Riak, and then expose GET requests on the image bucket to serve images directly from Riak, without channeling the requests through a web server.

<https://wiki.basho.com/display/RIAK/REST+API>

In contrast, Cassandra uses Thrift, which means that all access must be through a Thrift-based client.

<http://wiki.apache.org/cassandra/API>

Configurable Storage

Riak allows you to choose different storage backends for different buckets. If your application needs to store both small, frequently-accessed user profiles, and large, less-frequently-accessed MP3 files, you can configure Riak to store these things in a way that optimizes performance and accounts for business tradeoffs. You might store three replicas of the profile in our fast, Innostore-based backend, while storing only one copy of the .mp3 in a filesystem-based backend. This makes Riak appropriate for a wide variety of data.

<http://blog.basho.com/2010/02/02/basho-podcast-three---an-introduction-to-innostore/>

In contrast, as of version 0.5, Cassandra has only one setting for the number of replicas (ReplicationFactor), and persists all data using SSTables. (Cassandra 0.6, currently in beta, will support setting ReplicationFactor at a Keyspace level.)

<http://wiki.apache.org/cassandra/StorageConfiguration>

<http://wiki.apache.org/cassandra/ArchitectureSSTable>

Riak Compared to MongoDB

This is intended to be a brief, objective and technical comparison of Riak and MongoDB.

- [High Level Differences](#)
- [Replication and Scaling Out](#)
- [Scaling Back In](#)
- [Performance](#)
- [Data Model](#)
- [Queries and Distributed Operations](#)
- [Deflecting Conflicting Writes](#)
- [API](#)

High Level Differences

Riak and MongoDB satisfy some of the same use-cases :

- Semi-structured data modeled as "documents"
- Storage of non-document data in the database
- High write-availability

However, they approach solving these problems quite differently. Fundamentally, Riak is a distributed system while MongoDB is a single-system database (with support for replication and sharding). MongoDB specifies the internal format of documents while Riak is content-agnostic. MongoDB uses an ancillary specification called "GridFS" for storing non-document data, while Riak stores non-document or binary data in the same manner as structured data. MongoDB achieves high write-availability through performant in-place writes and "upserts", while Riak achieves availability through quorum writes, tolerance of node failure and hinted handoff.

<http://www.mongodb.org/display/DOCS/Home>

<http://blog.mongodb.org/post/248614779/fast-updates-with-mongodb-update-in-place>

<http://www.mongodb.org/display/DOCS/Updating#Updating-Update>

Replication and Scaling Out

Riak uses "consistent hashing" to replicate data. This functionality is deeply and tightly integrated with the Riak core, and enables Riak automatically replicate and rebalance data according to cluster size. Riak has no privileged nodes (no concept of master), making the system resilient to failure. Thus, when you need to grow your cluster for greater throughput or fault-tolerance, you simply add nodes.

[Add Nodes to Riak](#)

[Consistent Hashing](#)

Mongo's sharding is currently an alpha-release feature, and resembles the sharding design in Google's BigTable or Yahoo's PNUTS. Similarly to many relational databases, Mongo's replication has a concept of master and slave, but can also be set up in a paired-replicas configuration. Most production deployments of MongoDB have at least replication enabled, if not sharding as well.

<http://www.mongodb.org/display/DOCS/Sharding>

<http://www.mongodb.org/display/DOCS/Replication>

Scaling Back In

Riak allows you to remove a node from the cluster. When you do this, you hand the partitions that it manages back to the machines still in the cluster, giving each remaining node a few more partitions to worry about. In other words, the load remains evenly distributed.

At the time of writing, we could not find information on how to remove a MongoDB shard from a sharding setup.

Performance

Riak has pluggable storage engines, with the recommended being embedded InnoDB (the storage engine behind MySQL), so you can tune levels of performance and durability based on your needs. (One thing that is said around the office: "Eventual consistency is no excuse for losing data.") Durability and performance can also be tuned at the request level by specifying the number of nodes that need to agree on reads and writes.

Mongo is more performant because it uses memory-mapped files. The tradeoff is that it fsyncs (flushes in-memory data to disk) only every 60 seconds by default, so you run the risk of losing data if your MongoDB server goes down. The solution for increasing durability in MongoDB is to replicate.

<http://www.mongodb.org/display/DOCS/Durability+and+Repair>
<http://blog.mongodb.org/post/381927266/what-about-durability>

Data Model

Riak is content-type agnostic, and allows you to store semi-structured documents or objects of varying sizes. Riak lets you specify relationships between objects via links.

Data Storage in Riak

Mongo's data format is BSON (binary equivalent to JSON). Mongo stores this data as documents (self-contained records with no intrinsic relationships). Documents in MongoDB may store any of the defined BSON types, including raw binary data (upon which the GridFS feature is built).

<http://www.mongodb.org/display/DOCS/BSON>
<http://www.mongodb.org/display/DOCS/Data+Types+and+Conventions><http://www.mongodb.org/display/DOCS/GridFS>

Queries and Distributed Operations

Riak's query interface is entirely key-value, link-walking, or MapReduce. Riak has no concept of secondary indexes because it does not know the internal structure of the stored data.

<https://wiki.basho.com/display/RIAK/MapReduce>

MongoDB has a query interface that has some similarities to relational databases, including secondary indexes that can be derived from the stored documents. MongoDB also has a facility for performing MapReduce queries.

<http://www.mongodb.org/display/DOCS/Indexes>
<http://www.mongodb.org/display/DOCS/Querying><http://www.mongodb.org/display/DOCS/MapReduce>

Deflecting Conflicting Writes

Riak uses vector-clocks to track the ancestry of each object so your application can decide which representation to pick when two people have updated an object at the same time.

<http://riak.basho.com/edoc/vclock.html>

Mongo uses a "last one wins" technique for conflict resolution.

<http://www.mongodb.org/display/DOCS/Atomic+Operations>

API

Riak's primary interface to non-Erlang clients is via HTTP.

<https://wiki.basho.com/display/RIAK/REST+API>

Mongo uses a custom protocol with BSON as the interchange format, and 10gen supports clients in the most popular programming languages.

<http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol>

Riak Compared to Neo4j

This is intended to be a brief, objective and technical comparison of Riak and Neo4j.

- High Level Differences
- Scalability
- Data Model
- Conflicting Writes
- Querying

High Level Differences

Riak and Neo4j are meant for storing fundamentally different types of data:

- Riak is a document database and key/value store, designed to store semi-structured documents or objects of varying sizes.
- Neo4j is a graph database, designed to store and traverse a network of related information. (e.g. Social Networks)

In most cases, the needs of your application will clearly dictate whether you should use a key/value store or a graph database. And in many cases, it may make sense to combine the two. An application like Facebook, for example, might store user profile information, wall posts, and images in a key/value or document database, and store the network of friends and associations in a graph database.

Scalability

Riak was built to scale elastically, meaning that you can scale your cluster from one node to 100 nodes and beyond simply and easily. As you add nodes to your cluster, Riak automatically takes care of redistributing an equal share of the load to each server in your cluster. Likewise, if you scale your cluster down in size, Riak takes care of re-apportioning the data from the removed node evenly to the remaining nodes.

Adding Nodes to Riak

In contrast, Neo4j was designed to run on one machine and contains no built-in support to scale to multiple machines. That's not to say that you can't scale to multiple machines, it just means that your application must create its own sharding layer, and be smart enough to cleanly divide the data, which is a challenge, as graph databases generally store randomly connected webs of data. If data is not cleanly shardable, and instead is duplicated across multiple machines, then the sharding layer must be smart enough to coordinate Neo4j transactions, as Neo4j transactions are currently bound to a single machine.

<http://lists.neo4j.org/pipermail/user/2009-January/000997.html>
http://en.wikipedia.org/wiki/Six_degrees_of_separation

Data Model

Riak allows you to store semi-structured documents or objects of varying sizes. Riak is equally adept at storing a user profile, an image, an .mp3, a purchase order, or session information for a website.

Data Storage in Riak

In contrast, Neo4j stores data using nodes, relationships (imagine a line connecting the nodes), and properties. You can associate a list of properties on the node and the relationship. Properties are limited to Java primitives (int, byte, float, etc.), Strings, or an array of primitives and Strings. Relationships are typed, allowing you to express things like "PersonA KNOWS PersonB" and "PersonA IS_RELATED_TO PersonC".

<http://api.neo4j.org/current/org/neo4j/graphdb/PropertyContainer.html>

Conflicting Writes

Riak can detect when two processes try to update the same data with conflicting information by means of a vector clock. In a distributed environment, this happens more often than you would think: a client may update a cached version of an object, or a network split may have caused a client to delay its write. Riak can detect both of these cases, and uses the vector clock to determine which update should win, or to bubble the conflicting versions (called siblings) up to the client, where the application can choose which version wins, often with input from the user. (Think of what happens when two people edit a Wiki at the same time.)

<http://riak.basho.com/edoc/vclock.html>

In contrast, Neo4j supports configurable ACID transactions, similar to a traditional RDBMS. This allows a client to update a section of the graph in an isolated environment, hiding changes from other processes until the transaction is committed. If multiple transactions try to modify the same data, the Neo4j kernel will try to synchronize them. If interdependencies between the transactions would cause a deadlock, this will be detected and a corresponding exception will be thrown.

<http://wiki.neo4j.org/content/Transactions>
<http://wiki.neo4j.org/content/FAQ>

Riak's approach ensures that the datastore is always write-available, and that writes always succeed, even in the face of a network split or hardware failure, so long as the client can reach at least one node in the cluster. The tradeoff is that the client performing the read must do a little extra work to resolve the conflict, or can optionally choose to take the latest version of the object (this is the default setting.)

Neo4j's approach prevents conflicts from happening in the first place. The tradeoff is that client performing the write must do a little extra work to detect and retry a failed transaction, and, as previously mentioned, the transaction can only affect data on a single machine.

Querying

Riak allows you to access your data using a simple key/value model. In addition, Riak supports links and Javascript-based Map/Reduce:

- With links, you create lightweight pointers between your data, for example, from 'projects' to 'milestones' to 'tasks', and then select data along that hierarchy using simple client API commands. (In a pinch, this can substitute as a lightweight graph database, as long as the number of links is kept reasonably low; think dozens, not thousands.)

- With Javascript-based MapReduce you can define a MapReduce operation with multiple map and reduce phases, and then pass the operation to a Riak cluster along with a set of starting keys. Riak works like a mini-Hadoop, running the MapReduce operation in real time and returning the results. Any data processing happens in parallel across all machines, and any operations that run on a specific piece of data run on the machine that holds that data.

<https://wiki.basho.com/display/RIAK/MapReduce>

Neo4j, on the other hand, excels at querying networks of information. Again, drawing from Facebook, a graph database would make short work of finding all of the people who are friends of your friends. In relational parlance, if your queries start on a single row and explode into thousands of rows via recursive joins, then those relations should likely be stored in a graph database.

Neo4J requires you to provide a starting node before you can perform any queries or traversals. The starting node can be the result of a previous traversal, or may be retrieved by using the integer ID of the node generated by Neo4j. In this latter case, an application needs some way to map a real world value, such as a username, to a node ID. Neo4j currently supports tight integration with Lucene for this purpose, with support for ACID transactions on operations that touch both Neo4j and Lucene. Other than Lucene, any JTA compliant XA resource can participate in Neo4j transactions.

http://wiki.neo4j.org/content/Design_Guide

<http://highscalability.com/neo4j-graph-database-kicks-butt>

FAQ

- [Why did you build Yet Another Data Store?](#)
- [How do you pronounce "Riak"?](#)
- [How fast is it?](#)
- [But what about \(insert favorite "nosql" system here\)?](#)
- [Do I have to know how to write distributed Erlang code to use Riak?](#)
- [Is Version 0.x ready for my production data?](#)
- [What type of data can I store in Riak?](#)
- [What operating systems are best for Riak?](#)
- [How large of a file can one store in Riak? Any guidelines or recommendations?](#)
- [I wrote a great piece of code for Riak. How do I contribute it?](#)
- [What is the maximum number of partitions allowable in a Riak cluster?](#)
- [What is the largest cluster currently running?](#)
- [Why did you chose to support JavaScript \(in addition to Erlang\) for MapReduce?](#)
- [Can I delete an entire bucket?](#)
- [How can I authenticate users who are accessing Riak?](#)

Why did you build Yet Another Data Store?

We built the first incarnation of Riak in late 2007 for our own use, and we have nurtured it ever since in order to satisfy our operational needs. There wasn't an alternative that suited us at the time, and since then we have developed something that solves a unique combination of problems.

How do you pronounce "Riak"?

REE-ahk

How fast is it?

Performance depends on many factors, including hardware and network parameters as well as a great many tunable parameters in the way you set up your cluster and the way that you use Riak from an application. We've found it to be fast enough for our purposes, and our goal is not to be "fastest" but rather to stay "fast enough" as the system grows, as hosts fail, and so on. That said, as soon as we get a chance to produce a general, reproducible benchmarking suite, we'll share it with you.

But what about (insert favorite "nosql" system here)?

We believe that the current explosion of choice in data storage software is a healthy thing. Each of the different datastores out there has its own strengths and weaknesses and will be best suited to a different class of problems. We hope to not only contribute a strong system that works well for us and others, but also to work with friends in the open source community to improve the state of the art.

Do I have to know how to write distributed Erlang code to use Riak?

I believe you don't, Bob. Riak provides a simple and convenient programming interface over HTTP and JSON. This, combined with our [rapidly expanding suite of client libraries](#), makes Riak accessible to nearly every programmer on the Web.

Is Version 0.x ready for my production data?

That kind of decision depends on many factors, most of which cannot be answered in general but depend on your business. We gave it a low

version number befitting a first public appearance, but Riak has robustly served the needs of multiple revenue-generating applications for nearly two years now.

What type of data can I store in Riak?

Riak stores value as opaque BLOBS (binary large objects). This means that virtually any type of data is suitable for Riak.

What operating systems are best for Riak?

Riak is currently well-suited for Mac OS X and nearly all flavors of Linux. There is currently no support for Riak on Windows.

How large of a file can one store in Riak? Any guidelines or recommendations?

At the moment, the largest file size we recommend for storage in Riak is 50 MB. However, other factors may limit the size of your upload, including the number of replicas and the backend being used. For example, if you use the default `n_val` of 3, you will need RAM over 3 times the size of your largest upload, not including any RAM claimed by the backend. If you need to store really large files, consider breaking them up into smaller pieces. (There is support for larger files (in the multiple GB range) planned, but the date for this is yet to be determined.)

I wrote a great piece of code for Riak. How do I contribute it?

Please email the patch to the [riak-users](#) list or send a [pull request](#) via [Bitbucket](#) and we will take a look at it. (Note: you do need to have an existing Bitbucket account for the pull request to work.)

What is the maximum number of partitions allowable in a Riak cluster?

Theoretically, the maximum number of partitions is 2^{160} , the size of the consistent-hashing keyspace, but that number is not practical. We recommend around 10 partitions per physical node, but up to 50 is acceptable. The default number of partitions is 64. For larger clusters, choose 512 or 1024 (keeping in powers of 2).

- [Learn more about setting up your cluster](#)

What is the largest cluster currently running?

The most computers (nodes) in a single cluster that we have seen is about 60. The software performed as expected from the first through sixtieth hosts, and we expect that it would continue to do so with even larger clusters.

Why did you chose to support JavaScript (in addition to Erlang) for MapReduce?

JavaScript hit enough sweet spots that it made sense to start there: trivial dynamic loading of code, VMs meant for embedding, native handling of a common serialization format (JSON), and wide-spread developer familiarity (among other things). That said, support for other languages in MapReduce queries has not been ruled out at all. In fact, the attachment points for evaluating MapReduce functions are simple enough that we'd expect a determined community member could probably put together a good patch to support a new one.

Can I delete an entire bucket?

There is no standard operation for deleting a bucket. (Remember, buckets are essentially a flat namespace in Riak and have little significance beyond their ability to allow the same key name to exist in multiple buckets and to provide some per-bucket configurability.) Though it can be a costly operation, you can do a list keys operation on the bucket and delete all the keys individually.

How can I authenticate users who are accessing Riak?

Riak has no built-in authentication or authorization mechanism, as it was designed to be used in private networks and to be simple to configure. If you are using the [REST API](#), we recommend inserting a reverse-proxy between Riak and the client that can handle the authentication. Members of the community have used [nginx](#) and [Apache](#) for this purpose.

Recommended Resources

General Overviews, Tutorials, and Introductions

- [Justin Sheehy's Riak Presentation at NoSQL East](#)
- [Bryan Fink's Riak Presentation at NoSQL NYC](#)
- [Rusty Klophaus' Riak Presentation at NoSQL Live 2010](#)
- [Kevin Smith's "Introducing Riak" Presentation](#)
- [Riak and Nitrogen by Example - Rusty Klophaus at Erlang User Conf. 2009](#)
- [Riak in Ten Minutes- Jon Meredith at GlueCon 2010](#)
- [Why Riak Should Power Your Next Rails App](#)

- [Wrap Your Head Around Riak's MapReduce](#)
- [Link Walking By Example](#)
- [Why Vector Clocks Are Easy](#)
- [Riak and Ripple](#)
- [An Introduction to Innostore](#)
- [Schema Design in Riak - An Introduction](#)
- [Schema Design in Riak Pt. 2 - Relationships](#)

Sample Applications built using Riak:

- [Slideblast](#) - share and control slide presentation for the web.
- [riak_php_app](#) - a small PHP app that shows some basic usage of the Riak PHP library.
- [riak-url-shortener](#) - a small Ruby app (with Sinatra) that creates short URLs and stores them in Riak.
- [yakriak](#) - Riak-powered Ajax-polling chatroom.

Screencasts:

- [Introduction to JavaScript MapReduce in Riak](#)
- [Riak in an Embedded Node Using Rebar](#)
- [Setting up a Three Node Riak Cluster \(Used as part of The Riak Fast Track\)](#)
- [Brief Overview of JavaScript MapReduce in Riak \(Used as part of The Riak Fast Track\)](#)
- [Tuning CAP Controls in Riak \(Used as part of The Riak Fast Track\)](#)

Influential Papers, Books, and other Publications

- [Amazon's Dynamo Paper](#)
- [Distributed Systems: Principles and Paradigms](#)
- [Time, Clocks, and the Ordering of Events in a Distributed System](#)
- [Distributed Algorithms](#)
- [Elements of Distributed Computing](#)
- [Towards Robust Distributed Systems](#)
- [Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services](#)

Riak Glossary

Below is a list of terms and their applicability within the context of Riak:

- [Bucket](#)
- [Consistent Hashing](#)
- [Gossiping](#)
- [Hinted Handoff](#)
- [Key](#)
- [Links](#)
- [Merkle Tree](#)
- [Node](#)
- [Object](#)
- [Partition](#)
- [Read Repair](#)
- [Replica](#)
- [Ring](#)
- [Value](#)
- [Vector Clock](#)
- [Vnode](#)

Bucket

A Bucket is a container and [keyspace](#) for data stored in Riak, with a set of common properties for its contents (the number of [replicas](#), for instance). Buckets are accessed at the top of the URL hierarchy under "riak", e.g. `/riak/bucket`.

- [Take a more in depth look at Bucket Operations](#)

Consistent Hashing

Consistent hashing is a technique used to limit the reshuffling of keys when a hash-table data structure is rebalanced (when slots are added or removed). Riak uses consistent hashing to organize its data storage and replication. A Riak cluster is a 160-bit integer space which is divided into equally-sized partitions. Each [vnode](#) in the [Riak ring](#) is responsible for one of these partitions, and the location on the ring responsible for storing each object in Riak is determined using the consistent hashing technique.

- [More on Consistent Hashing](#)

Gossiping

Riak uses a "gossip protocol" to share and communicate ring state and bucket properties around the cluster. Whenever a node changes its claim on the ring, it announces its change via this protocol. Each node also periodically sends its current view of the ring state to a randomly-selected peer, in case any nodes missed previous updates.

Hinted Handoff

Hinted handoff is a technique for dealing with node failure in the Riak cluster in which neighboring nodes temporarily takeover storage operations for the failed node. When the failed node returns to the cluster, the updates received by the neighboring nodes are handed off to it.

Hinted handoff allows Riak to ensure database availability. When a node fails, Riak can continue to handle requests as if the node were still there.

Key

Keys are unique object identifiers in Riak and are scoped within [buckets](#).

Links

Links are metadata attached to objects in Riak. These links make establishing relationships between objects in Riak as simple as adding a Link header to the request when storing the object.

- [More about Riak's Links](#)
- [More on the Link header](#)

Merkle Tree

Merkle Trees are used in Riak to determine which objects on a node need updating. Nodes will exchange a Merkle Tree to determine what has changed instead of comparing every object on every node.

- [More information on Merkle Trees](#)

Node

A node is analogous to a physical server. Nodes run a certain number of [vnodes](#), which in turn claim a set number of [partitions](#) in the Riak ring key space.

Object

An object is another name for a [Value](#).

Partition

Partitions are the spaces into which a Riak cluster is divided. Each vnode in Riak is responsible for a partition, and data will be stored on a set number of partitions determined statically based on an object's key.

Read Repair

Read repair is an anti-entropy mechanism used by Riak to optimistically update stale replicas when they reply to a read request with stale data.

- [More about Read Repair](#)

Replica

Replicas are copies of data stored in Riak. The number of replicas required for both successful reads and writes is configurable in Riak and should be set based on your application's consistency and availability requirements.

Ring

The Riak Ring is a 160-bit integer space. This space is equally divided into [partitions](#), which are claimed by [vnodes](#), which reside on actual physical server [nodes](#).

Value

Riak is most-easily described as a key/value store. In Riak, "values" are opaque BLOBS (binary large objects), identified with a unique key, that can be any type of data, though some programming advantages are gained by using JSON.

Vector Clock

Riak utilizes vector clocks (short: `vclock`) to handle version control. Since any node in a Riak cluster is able to handle a request, and not all nodes need to participate, data versioning is required to keep track of a current value. When a value is stored in Riak, it is tagged with a vector clock and establishes the initial version. When it is updated, the client provides the vector clock of the object being modified so that this vector clock can be extended to reflect the update. Riak can then compare vector clocks on different versions of the object and determine certain attributes of the data.

- [Read more on Vector Clocks](#)
- [Blog post "Why Vector Clocks are Easy"](#)

Vnode

Vnodes, or "virtual nodes" are responsible for claiming a [partition](#) in the Riak [Ring](#), and they coordinate requests for these partitions. Vnodes are run on physical [nodes](#) in a Riak cluster, and the number of vnodes on each node is determined by the number of active physical nodes in the cluster.

Sample Data

When learning how to use a database, reading tutorials and blogs and watching videos will only take a user so far. At some point, you actually have to use the database! And, essential to using any database is having sample data to put into it.

This page is intended to be a repository of sample data for that exact purpose. We will be expanding it as often as possible with different types of data that are suitable for different types of application testing.

And, if you have sample data that you can spare or pointers to some data of which you are particularly fond, shoot an email to wiki@basho.com and we'll add it to the page and expand the list.

Sample Data

Google Historical Stock Data

Name	Size	Creator	Creation Date	Comment
 goog.csv	76 kB	Mark Phillips	Jun 06, 2010 22:15	

This `.csv` of GOOG historical stock data was originally used for [MapReduce examples in The Riak Fast Track](#). If you are looking for

Compressed Twitter Data

Need some Twitter data to play with? Linked [at the bottom of this great blog post by the Rowfeeder team](#) are a few compressed `.csv` files (coming in around 58MB) that might be appropriate for your testing.

Enron Email Dataset

What better to test with than the emails of corrupt corporate officials? [Hosted here courtesy of the Carnegie Mellon CS Department](#), the complete Enron Email Corpus is about 0.5M messages from around 150 different users, organized into folders for your testing convenience.

WindyCityDB NoSQL Lab Sample Data

The organizers at WindyCityDB aggregated some sample data to use with their lab session on NoSQL DBs. [Linked here](#) are some JSON and XML files that might be just what you need to take Riak for spin.

Best Practices

Innstore Configuration and Tuning

Some generally acceptable settings and practices that should provide good performance and reliability when using Innstore as a backend.

Network Security and Firewall Configurations

Covers some standard configurations and port settings to use when thinking about how to secure your Riak Cluster.

Hosting and Server Configuration

These are some recommendations for how to design and configure your hosting setup for Riak.

- [Operating System](#)
- [Hardware](#)
- [Virtualization](#)
- [Network configuration and Load-Balancing](#)

Operating System

We recommend using a mainstream Unix-like operating system on which to deploy Riak. Mainstream distributions have larger support communities and thus solutions for common problems are easier to find. Additionally, Basho builds binary packages of Riak for several mainstream distributions, easing deployment. Some acceptable distributions are:

- **Redhat based:** Redhat Enterprise Linux, CentOS, Fedora Core
- **Debian based:** Debian, Ubuntu
- **Solaris based:** Sun Solaris, OpenSolaris

Hardware

Riak is designed to scale horizontally – that is, to improve performance as you add nodes – but it can always take advantage of more powerful hardware. These are some generic recommendations for hardware configuration:

Multi-core 64-bit CPU - Because Riak is built on Erlang, more cores means more concurrency and thus greater performance. 64-bit architectures let Riak do certain numerical computations more efficiently.

Multiple GB RAM - More RAM means more data can be held in main memory, resulting in greater read, write, and [MapReduce](#) performance. Using too much RAM will increase swap utilization, causing performance to tank as memory operations begin to contend with normal disk operations.

Multiple fast hard disks (RAID and/or SSD) - Because many operations in Riak are I/O-bound, it is important to have fast hard disks to achieve good performance. It may help to configure the disks in RAID0 for increased read/write performance, and some backends like [Innstore](#) benefit directly from independent disks.

Fast network (Gigabit +) - Riak uses the network heavily for storage operations and for cluster status (ring-state gossip, handoff, etc). Fast interconnections between nodes and between clients and the cluster will improve performance.

Virtualization

Like most datastores, **Riak will run best when not virtualized**. Virtual machines (VMs) can suffer from poor I/O and network performance, depending on how they are configured and the environment in which they run. That said, here are some recommendations for running Riak in VPS or cloud environments:

Choose the largest VM you can afford. As mentioned in the [Hardware](#) section, better hardware means better performance. Additionally, larger instances are less likely to share hardware resources with other customers' virtual machines.

Deploy VMs within the same datacenter or region where possible. Some hosting providers allow you to choose the location of your servers when provisioning. Choosing to provision within the same datacenter or region will usually reduce network latency and increase throughput, resulting in greater performance.

Network configuration and Load-Balancing

There are at least two acceptable strategies for load-balancing requests across your Riak cluster, **virtual IPs** and **reverse-proxy**.

For **virtual IPs**, we recommend using any of the various VIP implementations but we don't recommend VRRP behaviour for the VIP because you'll lose the benefit of spreading client query load to all nodes in a ring.

For **reverse-proxy** configurations (HTTP interface), any one of these should work adequately: haproxy, squid, varnish, nginx, lighttpd, and Apache.

Innstore Configuration and Tuning

Innstore is a powerful backend that works for many production scenarios, but it has many tuning parameters that can affect the performance of your Riak cluster. Here are some generally acceptable settings and practices that should provide good performance and reliability.

- Configuration settings
- Other tuning techniques
- Miscellaneous

Configuration settings

 All of the settings go in the `innostore` section of `app.config`. See [Configuration Files](#) for more information about `app.config`.

buffer_pool_size (integer, # of bytes)

Set this to be 60-80% of available RAM (after subtracting RAM consumed by other services). For example, on a 12GB machine you might set it to 8GB:

```
{buffer_pool_size, 8589934592} %% 8 * 1024 * 1024 * 1024
```

flush_method (string)

On Linux and other Unix-like platforms, setting this to "O_DIRECT" will bypass a layer of filesystem buffering provided by the operating system. It is generally not necessary since Innostore does its own buffering. Example:

```
{flush_method, "O_DIRECT"}
```

open_files (integer)

Innostore opens a file for each partition/bucket combination, plus several files for its binary logs. Each of these count against the total number of files any one program may have open. As a result, you may need to adjust this number up or down from its default to accommodate a lower limit, or more open buckets. The default is 300. Example:

```
{open_files, 100} %% accommodate a lower limit
```



ulimit

To adjust the open files limit in your operating system, use the `ulimit -n` command.

log_group_home_dir & data_home_dir

For best performance, we recommend putting Innostore's log data on a separate hard disk from the actual data. This will also make its data more resilient to hardware failure – corrupted writes to the data files can often be recovered from the binary logs. Example:

```
{log_group_home_dir, "/data/innodb-log"}, %% mounted from /dev/hda1
{data_home_dir, "/data/riak/innodb"} %% mounted from /dev/hdb1
```

innodb log amount and size

When storing binary objects or working with a cluster the number of log files as well as their size should be increased to handle the additional amount of data being stored in Innostore. The following error exemplifies the log messages that will be seen if your log setup can't cope with the amount of data.

```
InnoDB: ERROR: the age of the last checkpoint is 30209814,
InnoDB: which exceeds the log group capacity 30195303.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

To fix this issue the following log settings will work for most environments:

```
{log_files_in_group, 6}, % How many files you need – usually, 3 < x < 6
{log_file_size, 268435456}, % No bigger than 256MB – otherwise recovery takes too long
```

Other tuning techniques

noatime

Innstore is very aggressive at reading and writing files. As such, you can get a big speed boost by adding the `noatime` mounting option to `/etc/fstab` (or wherever your platform defines the mounted devices. This will disable the recording of the "last accessed time" for all files. Example:

```
/dev/sda5 /data ext3 noatime 1 1
/dev/sdb1 /data/inno-log ext3 noatime 1 2
```

Miscellaneous

Key Size

Maximum key size for Innstore is 255 bytes.

Network Security and Firewall Configurations

Here are some standard configurations and port settings to use when thinking about how to secure your Riak Cluster:

There are two classes of access control for Riak:

1. other Riak nodes in the ring
2. clients making use of the Riak ring

For both access groups, the settings you want are in `riak/etc/app.config`. The config directives you care about for client access all end in `"_ip"` and `"_port"`: `web_ip`, `web_port`, `pb_ip`, and `pb_port`. Make note of those and configure your firewall to incoming TCP access to those ports or IP and port combinations. The exceptions to this is the `handoff_ip` and `handoff_port` directives. Those are for communication between Riak nodes only.

Riak uses the Erlang distribution mechanism for most inter-node communication. Riak identifies other machines in the ring using Erlang identifiers(`<identifier>@<hostname or IP>`, i.e. `"riak@10.9.8.7"`). Erlang resolves these node identifiers to a TCP port on a given machine via the Erlang Port Mapper daemon(`epmd`) running on each machine in a ring. `epmd` listens on TCP port 4369 on the wildcard interface.

You can configure Riak to tell the Erlang interpreter (and thence `epmd`) to only use a limited range of ports in `riak/etc/app.config`. If you want to restrict the range of ports that Erlang and `epmd` will use for inter-Erlang node communication to 6000-7999, you can add the following lines to `riak/etc/app.config`:

```
{ kernel, [
    { inet_dist_listen_min, 6000 },
    { inet_dist_listen_max, 7999 }
]},
```

Then just configure your firewall to allow incoming access to TCP ports 6000 to 7999 from whichever network(s) contain your Riak nodes.

Riak nodes in ring need to be able to communicate freely with one another on the following ports:

- `epmd`'s listener: TCP:4369
- `handoff_port` listener: TCP:8099
- range of ports you configure in `app.config`

Riak clients need to be able to contact a at least one machine in a Riak ring on the following ports:

- `web_port`: TCP:8098
- `pb_port`: TCP:8097

One important note: if you do add the `inet_dist_listen_min` and `inet_dist_listen_max` entries to `riak/etc/app.config`, you need to kill off any running `epmd` so it will pick up the new settings. `epmd` will continue to run on a given machine even after all Erlang interpreters have exited.

Benchmarking with Basho Bench

Basho Bench is a benchmarking tool created to conduct accurate and repeatable performance tests and stress tests, and produce performance graphs.

Originally developed by Dave Smith (Dizzy) to benchmark Riak, Basho's key/value datastore, it exposes a pluggable driver interface and has been extended to serve as a benchmarking tool against a variety of projects. New drivers can be written in Erlang and are generally less than 200 lines of code.

- [Download](#)
- [Documentation](#)
 - [How does it work?](#)

- Installation
 - Prerequisites
 - Building from Source
- Usage
- Generating Benchmark Graphs
 - Prerequisites
 - Generating a Graphs
- Configuration
 - Global Config Settings
 - basho_bench_driver_riakclient Settings
 - basho_bench_driver_dets Settings
 - basho_bench_driver_http_raw Settings
- Custom Driver

Download

The main repository for basho_bench is http://bitbucket.org/basho/basho_bench/.

Documentation



Note on Documentation

What follows is primarily a reprint of what can be found in the basho_bench repository under `docs/Documentation.org`

How does it work?

When Basho Bench starts (`basho_bench.erl`), it reads the configuration (`basho_bench_config.erl`), creates a new results directory, then sets up the test. (`basho_bench_app.erl`/`basho_bench_sup.erl`)

During test setup, Basho Bench creates:

- One **log process** (`basho_bench_log.erl`). During startup, this creates a new `/log.txt/` file in the current results directory, to which output is logged at the specified logging level.
- One **stats process** (`basho_bench_stats.erl`). This receives notifications when an operation completes, plus the elapsed time of the operation, and stores it in a histogram. At regular intervals, the histograms are dumped to `/summary.csv/` as well as operation-specific latency CSVs (e.g. `/put_latencies.csv/` for the 'put' operation).
- **N workers**, where N is specified by the `/concurrent/` configuration setting. (`basho_bench_worker.erl`). The worker process wraps a driver module, specified by the `/driver/` configuration setting. The driver is randomly invoked using the distribution of operations as specified by the `/operation/` configuration setting. The rate at which the driver invokes operations is governed by the `/mode/` setting.

Once these processes have been created and initialized, Basho Bench sends a run command to all worker processes, causing them to begin the test. Each worker is initialized with a common seed value for random number generation to ensure that the generated workload is reproducible at a later date.

During the test, the workers repeatedly call `/driver:/run/4`, passing in the next operation to run, a keygen function, a valuegen function, and the last state of the driver. The worker process times the operation, and reports this to the stats process when the operation has completed.

Finally, once the test has been run for the duration specified in the config file, all workers and stats processes are terminated and the benchmark ends. The measured latency and throughput of the test can be found in `./tests/current/`. Previous results are in timestamped directories of the form `./tests/YYYYMMDD-HHMMSS/`.

Installation

Prerequisites

- Erlang R13B03 - <http://erlang.org/download.html>
- R - <http://www.r-project.org/> (for graphing)

Building from Source

Basho Bench is currently available as source code only. To get the latest code, clone the basho_bench repository:

```
: hg clone ssh://hg@bitbucket.org/basho/basho_bench
: cd basho_bench
: make
```

Usage

Run basho_bench:

```
./basho_bench myconfig.config
```

This will generate results in `/tests/current/`. You will need to create a configuration file. The recommended approach is to start from a file in the `/examples/` directory and modify settings using the `/Configuration/` section below for reference.

Note that currently you must run the basho_bench script from the directory where it was built to ensure that the necessary dependencies are available.

Generating Benchmark Graphs

The output of basho_bench can be used to create graphs showing:

- Throughput - Operations per second over the duration of the test.
- Latency at 99th percentile, 99.9th percentile and max latency for the selected operations.
- Median latency, mean latency, and 95th percentile latency for the selected operations.

Prerequisites

The R statistics language is needed to generate graphs.

- More information: <http://www.r-project.org/>.
- Download R: <http://cran.r-project.org/mirrors.html>

Follow the instructions for your platform to install R.

Generating a Graphs

To generate a benchmark graph against the current results, run:

```
: make results
```

This will create a results file in `/tests/current/summary.png/`.

You can also run this manually:

```
: priv/summary.r -i tests/current
```

Configuration

Basho Bench ships with a number of sample configuration files, available in the `/examples/` directory.

Global Config Settings

mode

The `mode` setting controls the rate at which workers invoke the `/driver:/run/4`

function with a new operation. There are two possible values:

- `max` :: generate as many ops per second as possible
- `{rate, N}` :: generate N ops per second, with exponentially distributed interarrival times.

Note that this setting is applied to each driver independently. For example, if `{rate, 5}` is used with 3 concurrent workers, `basho_bench` will be generating 15 (i.e. $5 * 3$) operations per second.

```
: % Run at max, ie: as quickly as possible.  
: {mode, max}
```

```
: % Run 15 operations per second.  
: {mode, {rate, 15}}
```

concurrent

The number of concurrent worker processes. The default is 3 worker processes.

```
: % Run 10 concurrent processes.  
: {concurrent, 10}
```

duration

The duration of the test, in minutes. The default is 5 minutes.

```
: % Run the test for one hour.  
: {duration, 60}
```

operations

The possible operations that the driver will run, plus their "weight" or likelihood of being run. Default is `{get, 4}`, `{put, 4}`, `{delete, 1}` which means that out of every 9 operations, 'get' will be called four times, 'put' will be called four times, and 'delete' will be called once, on average.

```
: % Run 80% gets, 20% puts.  
: {operations, [{get, 4}, {put, 1}]}
```

Operations are defined on a **per-driver** basis. Not all drivers will implement the "get"/"put" operations discussed above. Consult the driver source to determine the valid operations.

If a driver does not support a specified operation ("askdfput" in this example) you may see errors like:

```
: DEBUG:Driver basho_bench_driver_null crashed: {function_clause,  
: {{{{basho_bench_driver_null,run,  
: [asdfput,  
: #Fun<basho_bench_keygen.4.4674>,  
: #Fun<basho_bench_valgen.0.1334>,  
: undefined}}}},  
: {{{{basho_bench_worker,  
: worker_next_op,1}}}},  
: {{{{basho_bench_worker,  
: max_worker_run_loop,1}}}}}
```

driver

The module name of the driver that `basho_bench` will use to generate load. A driver may simply invoke code in-process (such as when measuring the performance of `innostore` or `DETS`) or may open network connections and generate load on a remote system (such as when testing a Riak server/cluster).

Available drivers include:

- `basho_bench_driver_http_raw` :: Uses Riak's HTTP interface to get/put/delete data on a Riak server

- `basho_bench_driver_riakc_pb` :: Uses Riak's Protocol Buffers interface to get/put/delete data on a Riak server
- `basho_bench_driver_riakclient` :: Uses Riak's Dist. Erlang interface to get/put/delete data on a Riak server
- `basho_bench_driver_bitcask` :: Directly invokes the Bitcask API
- `basho_bench_driver_dets` :: Directly invokes the DETS API
- `basho_bench_driver_innystore` :: Directly invokes the Innostore API

On invocation of the `/driver/:run/4` method, the driver may return one of the following results:

- `={ok, NewState}=` :: operation completed successfully
- `={error, Reason, NewState}=` :: operation failed but the driver can continue processing (i.e. recoverable error)
- `={stop, Reason}=` :: operation failed; driver can't/won't continue processing
- `={'EXIT', Reason}=` :: operation failed; driver crashed

code_paths

Some drivers need additional Erlang code in order to run. Specify the paths to this code using the `code_paths` configuration setting.

As noted previously, `basho_bench /must/` be run in the directory it was built, for dependency reasons. `code_paths` should include, minimally, a reference to "deps/stats" which is the library that `basho_bench` uses for various statistical purposes.

For example:

```
:{{{code_paths, [
: "deps/stats",
: "../riak_src/apps/riak_kv",
: "../riak_src/apps/riak_core"]}}}
```

key_generator

The generator function to use for creating keys. Generators are defined in `/basho_bench_keygen.erl`. Available generators include:

- `{sequential_int, MaxKey}` :: generates integers from 0..MaxKey in order and then stops the system. Note that each instance of this keygen is specific to a worker.
- `{sequential_int_bin, MaxKey}` :: same as above, but the result from the function is a 32-bit binary encoding of the integer.
- `{sequential_int_str, MaxKey}` :: same as `/sequential_int/`, but the result from the function is encoded as a string.
- `{uniform_int, MaxKey}` :: selects an integer from uniform distribution of 0..MaxKey. I.e. all integers are equally probable.
- `{uniform_int_bin, MaxKey}` :: same as above, but the result of the function is a 32-bit binary encoding of the integer.
- `{uniform_int_str, MaxKey}` :: same as `/uniform_int/`, but the result from the function is encoded as a string.
- `{pareto_int, MaxKey}` :: selects an integer from a Pareto distribution, such that 20% of the available keys get selected 80% of the time. Note that the current implementation of this generator MAY yield values larger than MaxKey due to the mathematical properties of the Pareto distribution.
- `{pareto_int_bin, MaxKey}` :: same as `/pareto_int/`, but the result from the function is a 32-bit binary encoding of the integer.

The default key generator is `={uniform_int, 100000}=`.

Examples:

```
: % Use a randomly selected integer between 1 and 10,000
```

```
: {key_generator, {uniform_int, 10000}}.
```

```
: % Use a randomly selected integer between 1 and 10,000, as binary.
```

```
: {key_generator, {uniform_int_bin, 10000}}.
```

```
: % Use a pareto distributed integer between 1 and 10,000; values < 2000
```

```
: % will be returned 80% of the time.
```

```
: {key_generator, {pareto_int, 10000}}.
```

value_generator

The generator function to use for creating values. Generators are defined in /basho_bench_valgen.erl/. Available generators include:

- {fixed_bin, Size}:: generates a random binary of Size bytes. Every binary is the same size, but varies in content.
- {exponential_bin, MinSize, Mean}:: generate a random binary which has an exponentially-distributed size. Most values will be approximately MinSize + Mean bytes in size, with a long-tail of larger values.

The default value generator is =(value_generator, {fixed_bin, 100})=.

Examples:

```
: % Generate a fixed size random binary of 512 bytes
```

```
: {value_generator, {fixed_bin, 512}}.
```

```
: % Generate a random binary whose size is exponentially distributed
```

```
: % starting at 1000 bytes and a mean of 2000 bytes
```

```
: {value_generator, {exponential_bin, 1000, 2000}}.
```

rng_seed

The initial random seed to use. This is explicitly seeded, rather than seeded from the current time, so that a test can be run in a predictable, repeatable fashion.

Default is =(rng_seed, {42, 23, 12})=.

```
: % Seed to {12, 34, 56}
```

```
: {rng_seed, {12, 34, 56}}.
```

log_level

The **log_level** setting determines which messages Basho Bench will log to the console and to disk.

Default level is **debug**.

Valid levels are:

- debug
- info
- warn
- error

report_interval

How often, in seconds, should the stats process write histogram data to disk. Default is 10 seconds.

test_dir

The directory in which to write result data. The default is /tests/.

basho_bench_driver_riakclient Settings

These configuration settings apply to the /basho_bench_driver_riakclient/ driver.

riakclient_nodes

List of Riak nodes to use for testing.

```
:{riakclient_nodes, ['riak1@127.0.0.1', 'riak2@127.0.0.1']}
```

riakclient_cookie

The Erlang cookie to use to connect to Riak clients. Default is 'riak'.

```
:{riakclient_cookie, riak}.
```

riakclient_mynode

The name of the local node. This is passed into `=net_kernel:start/1=` (http://erlang.org/doc/man/net_kernel.html).

```
:{riakclient_mynode, ['basha_bench@127.0.0.1', longnames]}.
```

riakclient_replies

This value is used for R-values during a get operation, and W-values during a put operation.

```
:% Expect 1 reply.  
:{riakclient_replies, 1}.
```

riakclient_bucket

The Riak bucket to use for reading and writing values. Default is `==<<"test">>=`.

```
:% Use the "bench" bucket.  
:{riakclient_bucket, <<"bench">>}.
```

basho_bench_driver_dets Settings

Not yet documented.

basho_bench_driver_http_raw Settings

http_raw_ips

List of IP addresses to connect the workers to. Each worker makes requests to each IP in a round-robin fashion.

```
Default is={http_raw_ips, "127.0.0.1"}=
```

```
% Connect to a cluster of machines in the 10.x network  
{http_raw_ips, "10.0.0.1", "10.0.0.2", "10.0.0.3"}.
```

http_raw_port

Select the default port to connect on for the HTTP server.

```
Default is={http_raw_port, 8098}=
```

```
% Connect on port 8090  
{http_raw_port, 8090}.
```

http_raw_path

Base path to use for accessing riak - usually `"/riak/<bucket>"`

```
Defaults is={http_raw_path, "/riak/test"}=
```

```
% Place test data in another_bucket  
{http_raw_path, "/riak/another_bucket"}.
```

http_raw_params

Additional parameters to add to the end of the URL. This can be used to set riak r/w/dw/rw parameters as as desired.

Default is `={http_raw_params, ""}`.

`% Set R=1, W=1 for testing a system with n_val set to 1`
`{http_raw_params, "?r=1&w=1"}.`

Custom Driver

A custom driver must expose the following callbacks.

`: % Create the worker.`

`: % ID is an integer.`

`: new(ID) -> {ok, State} or {error, Reason}.`

`: % Run an operation.`

`: run(Op, KeyGen, ValueGen, State) -> {ok, NewState} or {error, Reason, NewState}`

See the existing drivers for more details.

Riak Recaps

The Riak Recap is a daily(ish) email to the Riak Users Mailing List that summarizes any interesting Riak-related discussions and events from the previous day and also attempts to answer any questions from IRC, Twitter, etc. that may have gone unanswered.

They are listed here, with brief descriptions, for indexing and posterity purposes.

- 6/18/2010-6/21/2010: Loading Data in Riak; Hinted Writes in Riak; Memory Issues with Bitcask; Updated Python Client
- 6/16/2010-6/17/2010: Drop Table in Riak?; Riak as Distributed Filesystem; Advantages of Riak PB Client; Testing Riak
- 6/15/2010: Updated Riak Python Client; Net-Riak Driver; Riak MapReduce at LA Ruby
- 6/14/2010: Updated Riak Java Client; Clojure/Protocol Buffers Client; Embedding Riak into Applications
- 6/10/2010-6/13/2010: Riak Vector Clocks; Riak Testing; Vector Clock as a Tree?; Bitcask, Innostore and File Limits
- 6/9/2010: Basho Bench; Riak Chef Cookbook
- 6/8/2010: Pure Erlang Application using Riak; Riak over Cassandra; Riak Benchmarking Webinar
- 6/7/2010: Hosting and Configuration Best Practices; Dayfindr on Riak
- 6/4/2010-6/6/2010: HEAD requests in Bitcask; Cassandra or Riak; JSCouch-Riak; Link Queries
- 6/2/2010: Riak-js Enhancements; Deleting Embedded Documents via Ripple; Pre and Post Commit Hooks for Conflict Resolution
- 6/1/2010: Node Riak; Rolling with Riak
- 5/31/2010: Riak MapReduce Behavior; Bucket Parameter Replication; Single Node Riak Setups
- 5/28/2010-5/30/2010: Riak and Node.js; Joe Armstrong mentions Riak
- 5/26/2010-5/27/2010: Hinted Handoff, Partition Tolerance, Large Clusters; Riak's Memory Footprint
- 5/25/2010: Listing Buckets and Keys with MapReduce
- 5/24/2010: What's a vtag?
- 5/21/2010-5/23/2010: Ruby PB Client; Webmachine Resource on each Riak Node; Setting N, R, and W in Innostore
- 5/20/2010: Riak with ExtJS; Encryption on Disk; Using Round Robin Distribution to Query Riak
- 5/18/2010-5/19/2010: More work on Riak Fuse; Riak Disk Usage Stats
- 5/17/2010: Setting R Value Higher than N; Code for tearing down Riak in eunit test
- 5/14/2010-5/16/2010: List Keys; Riak Benchmarking, Innostore and Bitcask; GUI for Riak written in Java
- 5/12/2010-5/13/2010: Riakfuse; Erlang Interface or Protobuffs?; Replication Specifics; Porting MongoDB app to Riak
- 5/11/2010: Update Riak Metadata; Two Clusters on One Node
- 5/10/2010: Commit Hooks; Storing Images in Riak with Ripple
- 5/7/2010-5/9/2010: .NET; Node.js and Nori Bitcask Enhancements
- 5/5/2010-5/6/2010: Link Buckets in Riak; Data Migration; Riak Opscode Cookbook
- 5/4/2010: Riak_web_ip; \$PATH_TO_RIAK; Riak.js and Kiwi; The Riak Fast Track
- 5/3/2010: Riak MapReduce Performance and JavaScript
- 4/29/2010-5/2/2010: Riak and Node.js
- 4/28/2010: Riak-Admin Test; Riak and Etags; Riak and Ripple pre Rails 3
- 4/26/2010-4/27/2010: Riak for Large Objects; From One Node to Multiple Nodes; Exiting Protocol Buffers Process
- 4/24/2010-4/25/2010: Riak and Homebrew; Setting up Erlang and Webmachine on Rackspace
- 4/21/2010-4/22/2010: Removing Nodes from a Cluster; Protocol Buffers and Pre- and Post- Commit Hooks
- 4/18/2010-4/20/2010: Riak on Github
- 4/15/2010: Adding JSONP Support; Connection Two Nodes on AWS
- 4/14/2010: Cleaning up JavaScript VMs; Bucket Creation; MapReduce
- 4/12/2010-4/13/2010: Listing all Values through Erlang Client; Sean Cribbs Preso
- 4/9/2010-4/11/2010: Bucket Deletion; Using MapReduce for Dynamic Queries

- 4/5/2010-4/6/2010: Nodes joining clusters; MapReduce to limit results; Storing MapReduce Result; Link Walking
- 4/1/2010: JavaScript MapReduce; Using Bucket in MapReduce; NoSQL Write-up from @bkaney
- 3/31/2010: The first ever Riak Recap

Contact Basho

Need to get in touch with us? Here are the best ways to do it:

- To discuss Riak with the entire community, start by [subscribing to the mailing list](#).
- To contact the core Riak Development Team directly, send an email to riak@basho.com.
- For commercial enquiries, you can [read more Riak EnterpriseDS](#) or use the [contact form on Basho.com](#) to reach the sales team directly.
- If you think the Riak Wiki is missing something or you have other general comments about it, send an email to wiki@basho.com with your thoughts.