a little riak book

2.0

eric redmond john daily

basho

A Little Riak Book

Eric Redmond & John Daily*

2.0.0 2014-08-14

*Special thanks to everyone who helped and all of the folks at Basho

Body typeface is Crimson.

Contents

Chapter 1

Introduction

Downtime Roulette

Picture a roulette wheel in a casino, where any particular number has a 1 in 37 chance of being hit. Imagine you could place a single bet that a given number will *not* hit (about 97.3% in your favor), and winning would pay out 10 times your wager. Would you make that bet? I'd reach for my wallet so fast my thumb would start a fire on my pocket.

Now imagine you could bet again, but only win if the wheel made a sequential 100 spins in your favor, otherwise you lose. Would you still play? Winning a single bet might be easy, but over many trials the odds are not in your favor.

People make these sorts of bets with data all of the time. A single server has a good chance of remaining available. When you run a cluster with thousands of servers, or billions of requests, the odds of any one breaking down becomes the rule.

A once-in-a-million disaster is commonplace in light of a billion opportunities.

What is Riak

Riak is an open-source, distributed key/value database for high availability, fault-tolerance, and near-linear scalability. In short, Riak has remarkably high uptime and grows with you.

As the modern world stitches itself together with increasingly intricate connections, major shifts are occurring in information management. The web and networked devices spur an explosion of data collection and access unseen in the history of the world. The magnitude of values stored and managed continues to grow at a staggering rate, and in parallel, more people than ever require fast and reliable access to this data. This trend is known as *Big Data*.

So What is Big Data?

There's a lot of discussion around what constitutes Big Data.

I have a 6 Terabyte RAID in my house to store videos and other backups. Does that count? On the other hand, CERN grabbed about 200 Petabytes looking for the Higgs boson.

It's a hard number to pin down, because Big Data is a personal figure. What's big to one might be small to another. This is why many definitions don't refer to byte count at all, but instead about relative potentials. A reasonable, albeit wordy, definition of Big Data is provided by Gartner:

Big Data are high-volume, high-velocity, and/or high-variety information figures that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.

Always Bet on Riak

The sweet spot of Riak is high-volume (data that's available to read and write when you need it), high-velocity (easily responds to growth), and high-variety information figures (you can store any type of data as a value).

Riak was built as a solution to real Big Data problems, based on the *Amazon Dynamo* design. Dynamo is a highly available design—meaning that it responds to requests quickly at very large scales, even if your application is storing and serving terabytes of data a day. Riak had been used in production prior to being open-sourced in 2009. It's currently used by Github, Comcast, Voxer, Disqus and others, with the larger systems storing hundreds of TBs of data, and handling several GBs per node daily.

Riak was written on the Erlang programming language. Erlang was chosen due to its strong support for concurrency, solid distributed communication, hot code loading, and fault-tolerance. It runs on a virtual machine, so running Riak requires an Erlang installation.

So should you use Riak? A good rule of thumb for potential users is to ask yourself if every moment of downtime will cost you in some way (money, users, etc). Not all systems require such extreme amounts of uptime, and if you don't, Riak may not be for you.

About This Book

This is not an "install and follow along" guide. This is a "read and comprehend" guide. Don't feel compelled to have Riak, or even have a computer handy, when starting this book. You may feel like installing at some point, and if so, instructions can be found in the Riak docs.

In my opinion, the most important section of this book is the concepts chapter. If you already have a little knowledge it may start slow, but it picks up in a hurry. After laying the theoretical groundwork, we'll move onto helping developers use Riak, by learning how to query it and tinker with some settings. Finally, we'll go over the basic details that operators should know, such as how to set up a Riak cluster, configure some values, use optional tools, and more.

Changes From 1.x

Riak 2.0 represents a major shift in the capabilities and focus of Riak as a data store. Riak has always been primarily focused on operational simplicity, and that has not changed. But when it came to design decisions, operations were always given unilateral priority over the needs of developers. This is changing. With the launch of 2.0, we've finally added a few features that developers have wanted to see for quite a long time. Namely the following:

- **Strong Consistency**. Riak is still Eventually Consistent, too, but now you have a choice. Riak is now the easiest to manage database for adjusting the spectrum smoothly between AP and CP… per bucket, no less.
- **Better Search**. The makers of Riak have improved search by leveraging the power of the Solr search engine. You now get all of the queryability of distributed Solr, without the hassle of manual indexing.
- **Datatypes**. Riak historically has provided storage flexibility by allowing the storage of any binary object. This is still the case, but you now have the option of storing distributed maps, sets, counters, and flags that automatically converge in the face of conflicts.
- **Security**. A long standing request that's day has finally come. Native Group/User access controls.
- **Bucket types**. Now you can support unlimited custom bucket properties, without the overhead of the old gossip protocol.
- **Ring Resizing**. Finally! Where in the past you were limited to a fixed ring size, you now have the option to dynamically increase/decrease the number of vnodes in your cluster.
- **Other improvements**. We've also made many other improvements, like simplified configuration management (no more messing with `app.config` and `vm.args`), reduced sibling explosions (via a new logical clock called DVV), improved internal metadata sharing (reducing gossip chatter), better AAE, and more.

This book also includes a new chapter written by John Daily, to help guide developers to write productive applications with Riak. We hope you enjoy the new, improved, *Not Quite So Little Riak Book*.

Chapter 2

Concepts

Believe me, dear reader, when I suggest that thinking in a distributed fashion is awkward. When I had first encountered Riak, I was not prepared for some of its more preternatural concepts. Our brains just aren't hardwired to think in a distributed, asynchronous manner. Richard Dawkins coined the term *Middle World*—the serial, rote land humans encounter every day, which exists between the extremes of the very small strangeness of quarks and the vastness of outer space. We don't consider these extremes clearly because we don't encounter them on a daily basis, just like distributed computations and storage. So we create models and tools to bring the physical act of scattered parallel resources in line to our more ordinary synchronous terms. While Riak takes great pains to simplify the hard parts, it does not pretend that they don't exist. Just like you can never hope to program at an expert level without any knowledge of memory or CPU management, so too can you never safely develop a highly available cluster without a firm grasp of a few underlying concepts.

The Landscape

The existence of databases like Riak is the culmination of two basic trends: accessible technology spurring different data requirements, and gaps in the data management market.

First, as we've seen steady improvements in technology along with reductions in cost, vast amounts of computing power and storage are now within the grasp of nearly anyone. Along with our increasingly interconnected world caused by the web and shrinking, cheaper computers (like smartphones), this has catalyzed an exponential growth of data, and a demand for more predictability and speed by savvier users. In other words, more data is being created on the front-end, while more data is being managed on the backend.

Second, relational database management systems (RDBMS) have become focused over the years for a standard set of use-cases, like business intelligence. They were also technically tuned for squeezing performance out of single larger servers, like optimizing disk access, even while cheap commodity (and virtualized) servers made horizontal growth increasingly attractive. As cracks in relational implementations became apparent, custom implementations arose in response to specific problems not originally envisioned by the relational DBs.

These new databases are collected under the moniker *NoSQL*, and Riak is of its ilk.

Database Models

Modern databases can be loosely grouped into the ways they represent data. Although I'm presenting 5 major types (the last 4 are considered NoSQL models), these lines are often blurred—you can use some key/value stores as a document store, you can use a relational database to just store key/value data.

A Quick note on JOINs

Unlike relational databases, but similar to document and columnar stores, objects cannot be joined by Riak. Client code is responsible for accessing values and merging them, or by other code such as MapReduce.

The ability to easily join data across physical servers is a tradeoff that separates single node databases like relational and graph, from *naturally partitionable* systems like document, columnar, and key/value stores.

This limitation changes how you model data. Relational normalization (organizing data to reduce redundancy) exists for systems that can cheaply join data together per request. However, the ability to spread data across multiple nodes requires a denormalized approach, where some data is duplicated, and computed values may be stored for the sake of performance.

1. **Relational**. Traditional databases usually use SQL to model and query data. They are useful for data which can be stored in a highly structured schema, yet require flexible querying. Scaling a relational database (RDBMS) traditionally occurs by more powerful hardware (vertical growth).

   Examples: *PostgreSQL*, *MySQL*, *Oracle*

2. **Graph**. These exist for highly interconnected data. They excel in modeling complex relationships between nodes, and many implementations can handle multiple billions of nodes and relationships (or edges and vertices). I tend to include *triplestores* and *object DBs* as specialized variants.

   Examples: *Neo4j*, *Graphbase*, *InfiniteGraph*

3. **Document**. Document datastores model hierarchical values called documents, represented in formats such as JSON or XML, and do not enforce a document schema. They generally support distributing across multiple servers (horizontal growth).

   Examples: *CouchDB*, *MongoDB*, *Couchbase*

4. **Columnar**. Popularized by Google's BigTable, this form of database exists to scale across multiple servers, and groups similar data into column families. Column values can be individually versioned and managed, though families are defined in advance, not unlike RDBMS schemas.

   Examples: *HBase*, *Cassandra*, *BigTable*

5. **Key/Value**. Key/Value, or KV stores, are conceptually like hashtables, where values are stored and accessed by an immutable key. They range from single-server varieties like *Memcached* used for high-speed caching, to multi-datacenter distributed systems like *Riak Enterprise*.

   Examples: *Riak*, *Redis*, *Voldemort*

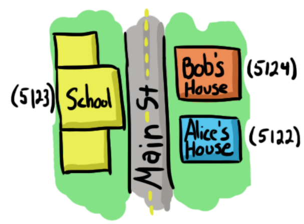
Riak Components

Riak is a Key/Value (KV) database, built from the ground up to safely distribute data across a cluster of physical servers, called nodes. A Riak cluster is also known as a ring (we'll cover why later).

Riak functions similarly to a very large hash space. Depending on your background, you may call it hashtable, a map, a dictionary, or an object. But the idea is the same: you store a value with an immutable key, and retrieve it later.

Key and Value

Key/value is the most basic construct in all of computerdom. You can think of a key like a home address, such as Bob's house with the unique key 5124, while the value would be maybe Bob (and his stuff).




```
hashtable["5124"] = "Bob"
```

Retrieving Bob is as easy as going to his house.

```
bob = hashtable["5124"]
```

Let's say that poor old Bob dies, and Claire moves into this house. The address remains the same, but the contents have changed.

```
hashtable["5124"] = "Claire"
```

Successive requests for `5124` will now return `Claire`.

Buckets

Addresses in Riakville are more than a house number, but also a street. There could be another 5124 on another street, so the way we can ensure a unique address is by requiring both, as in *5124 Main Street*.

Buckets in Riak are analogous to street names: they provide logical namespaces so that identical keys in different buckets will not conflict.

For example, while Alice may live at *5122 Main Street*, there may be a gas station at *5122 Bagshot Row*.

```
main["5122"] = "Alice"
bagshot["5122"] = "Gas"
```

Certainly you could have just named your keys `main_5122` and `bagshot_5122`, but buckets allow for cleaner key naming, and have other benefits, such as custom properties. For example, to add new Riak Search 2.0 indexes to a bucket, you might tell Riak to index all values under a bucket like this:

```
main.props = {"search_index":"homes"}
```

Buckets are so useful in Riak that all keys must belong to a bucket. There is no global namespace. The true definition of a unique key in Riak is actually `bucket/key`.

Bucket Types

Starting in Riak 2.0, there now exists a level above buckets, called bucket types. Bucket types are groups of buckets with a similar set of properties. So for the example above, it would be like a bucket of keys:

```
places["main"]["5122"] = "Alice"
places["bagshot"]["5122"] = "Gas"
```

The benefit here is that a group of distinct buckets can share properties.

```
places.props = {"search_index":"anyplace"}
```

This has practical implications. Previously, you were limited to how many custom bucket properties Riak could support, because any slight change from the default would have to be propogated to every other node in the cluster (via the gossip protocol). If you had ten thousand custom buckets, that's ten thousand values that were routinely sent amongst every member. Quickly, your system could be overloaded with that chatter, called a *gossip storm*.

With the addition of bucket types, and the improved communication mechanism that accompanies it, there's no limit to your bucket count. It also makes managing multiple buckets easier, since every bucket of a type inherits the common properties, you can make across-the-board changes trivially.

Due to its versatility (and downright necessity in some cases) and improved performance, Basho recommends using bucket types whenever possible from this point into the future.

For convenience, we call a *type/bucket/key + value* pair an *object*, sparing ourselves the verbosity of "X key in the Y bucket with the Z type, and its value".

Replication and Partitions

Distributing data across several nodes is how Riak is able to remain highly available, tolerating outages and network partitions. Riak combines two styles of distribution to achieve this: replication and partitions.

Replication

Replication is the act of duplicating data across multiple servers. Riak replicates by default.

The obvious benefit of replication is that if one node goes down, nodes that contain replicated data remain available to serve requests. In other words, the system remains *available*.

For example, imagine you have a list of country keys, whose values are those countries' capitals. If all you do is replicate that data to 2 servers, you would have 2 duplicate databases.

The downside with replication is that you are multiplying the amount of storage required for every duplicate. There is also some network overhead with this approach, since values must also be routed to all replicated nodes on write. But there is a more insidious problem with this approach, which I will cover shortly.

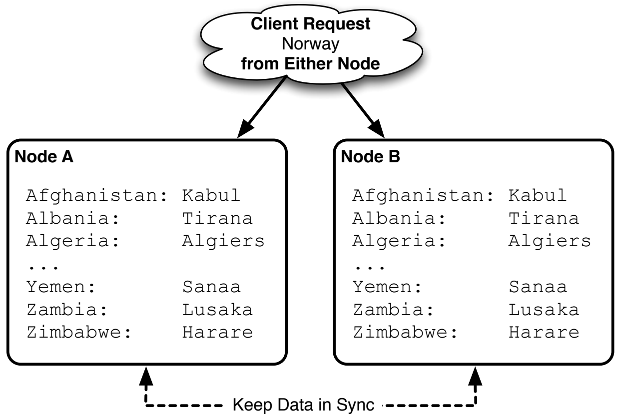


Figure 2.1: Replication

Partitions

A **partition** is how we divide a set of keys onto separate physical servers. Rather than duplicate values, we pick one server to exclusively host a range of keys, and the other servers to host remaining non-overlapping ranges.

With partitioning, our total capacity can increase without any big expensive hardware, just lots of cheap commodity servers. If we decided to partition our database into 1000 parts across 1000 nodes, we have (hypothetically) reduced the amount of work any particular server must do to 1/1000th.

For example, if we partition our countries into 2 servers, we might put all countries beginning with letters A-N into Node A, and O-Z into Node B.

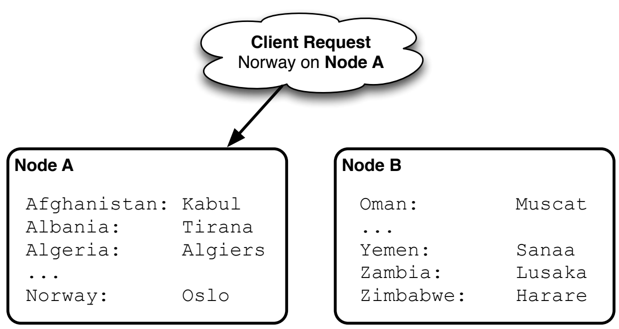


Figure 2.2: Partitions

There is a bit of overhead to the partition approach. Some service must keep track of what range of values live on which node. A requesting application must know that the key `Spain` will be routed to Node B, not Node A.

There's also another downside. Unlike replication, simple partitioning of data actually *decreases* uptime. If one node goes down, that entire partition of data is unavailable. This is why Riak uses both replication and partitioning.

Replication+Partitions

Since partitions allow us to increase capacity, and replication improves availability, Riak combines them. We partition data across multiple nodes, as well as replicate that data into multiple nodes.

Where our previous example partitioned data into 2 nodes, we can replicate each of those partitions into 2 more nodes, for a total of 4.

Our server count has increased, but so has our capacity and reliability. If you're designing a horizontally scalable system by partitioning data, you must deal with replicating those partitions.

The Riak team suggests a minimum of 5 nodes for a Riak cluster, and replicating to 3 nodes (this setting is called `n_val`, for the number of *nodes* on which to replicate each object).

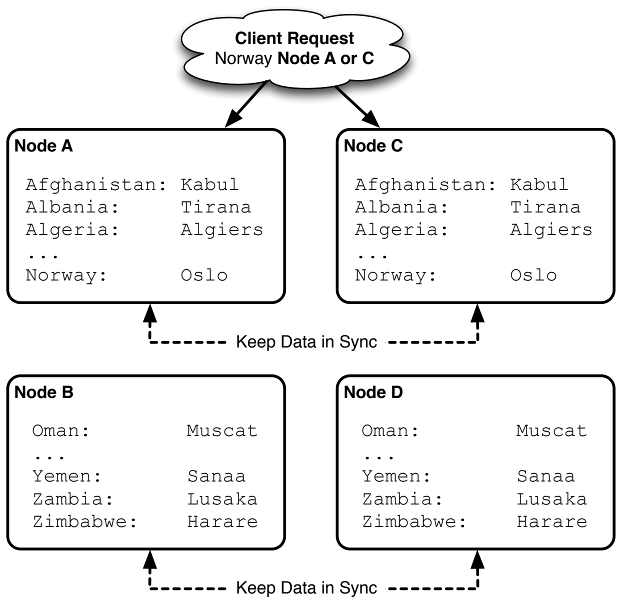


Figure 2.3: Replication Partitions

The Ring

Riak applies *consistent hashing* to map objects along the edge of a circle (the ring).

Riak partitions are not mapped alphabetically (as we used in the examples above), but instead a partition marks a range of key hashes (SHA-1 function applied to a key). The maximum hash value is 2^{160}, and divided into some number of partitions—64 partitions by default (the Riak config setting is `ring_creation_size`).

Let's walk through what all that means. If you have the key `favorite`, applying the SHA-1 algorithm would return `7501 7a36 ec07 fd4c 377a 0d2a 0114 00ab 193e 61db` in hexadecimal. With 64 partitions, each has 1/64 of the 2^{160} possible values, making the first partition range from 0 to 2^{154}-1, the second range is 2^{154} to 2*2^{154}-1, and so on, up to the last partition 63*2^{154}-1 to 2^{160}-1.

We won't do all of the math, but trust me when I say `favorite` falls within the range of partition 3.

If we visualize our 64 partitions as a ring, `favorite` falls here.

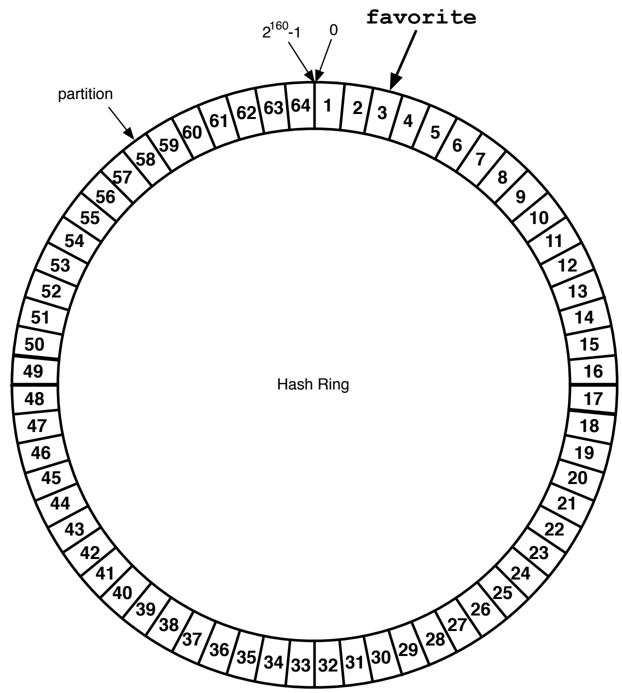


Figure 2.4: Riak Ring

"Didn't he say that Riak suggests a minimum of 5 nodes? How can we put 64 partitions on 5 nodes?" We just give each node more than one partition, each of which is managed by a *vnode*, or *virtual node*.

We count around the ring of vnodes in order, assigning each node to the next available vnode, until all vnodes are accounted for. So partition/vnode 1 would be owned by Node A, vnode 2 owned by Node B, up to vnode 5 owned by Node E. Then we continue by giving Node A vnode 6, Node B vnode 7, and so on, until our vnodes have been exhausted, leaving us this list.

- A = [1,6,11,16,21,26,31,36,41,46,51,56,61]
- B = [2,7,12,17,22,27,32,37,42,47,52,57,62]
- C = [3,8,13,18,23,28,33,38,43,48,53,58,63]
- D = [4,9,14,19,24,29,34,39,44,49,54,59,64]
- E = [5,10,15,20,25,30,35,40,45,50,55,60]

So far we've partitioned the ring, but what about replication? When we write a new value to Riak, it will replicate the result in some number of nodes, defined by a setting called `n_val`. In our 5 node cluster it defaults to 3.

So when we write our `favorite` object to vnode 3, it will be replicated to vnodes 4 and 5. This places the object in physical nodes C, D, and E. Once the write is complete, even if node C crashes, the value is still available on 2 other nodes. This is the secret of Riak's high availability.

We can visualize the Ring with its vnodes, managing nodes, and where `favorite` will go.

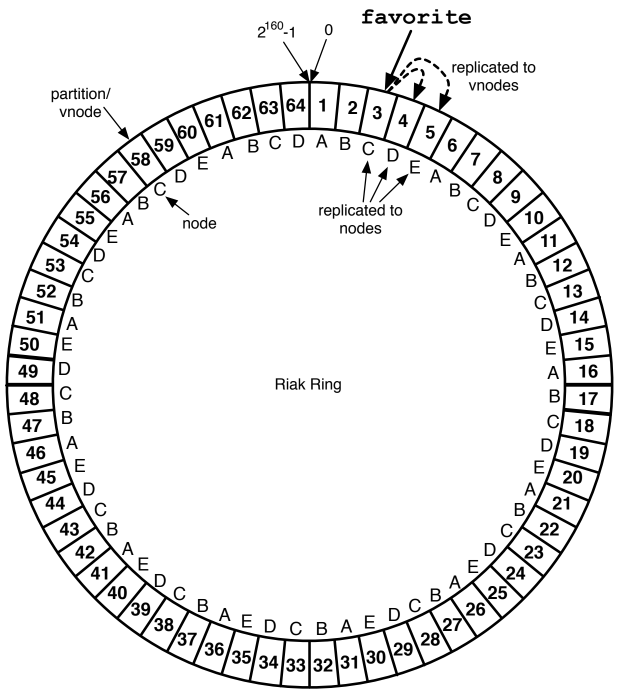


Figure 2.5: Riak Ring

The Ring is more than just a circular array of hash partitions. It's also a system of metadata that gets copied to every node. Each node is aware of every other node in the cluster, which nodes own which vnodes, and other system data.

Armed with this information, requests for data can target any node. It will horizontally access data from the proper nodes, and return the result.

Practical Tradeoffs

So far we've covered the good parts of partitioning and replication: highly available when responding to requests, and inexpensive capacity scaling on commodity hardware. With the clear benefits of horizontal scaling, why is it not more common?

CAP Theorem

Classic RDBMS databases are *write consistent*. Once a write is confirmed, successive reads are guaranteed to return the newest value. If I save the value `cold pizza` to my key `favorite`, every future read will consistently return `cold pizza` until I change it.

But when values are distributed, *consistency* might not be guaranteed. In the middle of an object's replication, two servers could have different results. When we update `favorite` to `cold pizza` on one node, another node might contain the older value `pizza`, because of a network connectivity problem. If you request the value of `favorite` on either side of a network partition, two different results could possibly be returned—the database is inconsistent.

If consistency should not be compromised in a distributed database, we can choose to sacrifice *availability* instead. We may, for instance, decide to lock the entire database during a write, and simply refuse to serve requests until that value has been replicated to all relevant nodes. Clients have to wait while their results can be brought into a consistent state (ensuring all replicas will return the same value) or fail if the nodes have trouble communicating. For many high-traffic read/write use-cases, like an online shopping cart where even minor delays will cause people to just shop elsewhere, this is not an acceptable sacrifice.

This tradeoff is known as Brewer's CAP theorem. CAP loosely states that you can have a C (consistent), A (available), or P (partition-tolerant) system, but you can only choose 2. Assuming your system is distributed, you're going to be partition-tolerant, meaning, that your network can tolerate packet loss. If a network partition occurs between nodes, your servers still run. So your only real choices are CP or AP. Riak 2.0 supports both modes.

Strong Consistency

Since version 2.0, Riak now supports strong Consistency (SC), as well as High Availability (HA). "Waitaminute!" I hear you say, "doesn't that break the CAP theorem?" Not the way Riak does it. Riak supports setting a bucket type property as strongly consistent. Any bucket of that type is now SC. Meaning, that a request is either successfully replicated to a majority of partitions, or it fails (if you want to sound fancy at parties, just say "Riak SC uses a variant of the vertical Paxos leader election algorithm").

This, naturally, comes at a cost. As we know from the CAP theorem, if too many nodes are down, the write will fail. You'll have to repair your node or network, and try the write again. In short, you've lost high availability. If you don't absolutely need strong consistency, consider staying with the high availability default, and tuning it to your needs as we'll see in the next section.

Tunable Availability with N/R/W

A question the CAP theorem demands you answer with a distributed system is: do I give up strong consistency, or give up ensured availability? If a request comes in, do I lock out requests until I can enforce consistency across the nodes? Or do I serve requests at all costs, with the caveat that the database may become inconsistent?

Riak's solution is based on Amazon Dynamo's novel approach of a *tunable* AP system. It takes advantage of the fact that, though the CAP theorem is true, you can choose what kind of tradeoffs you're willing to make. Riak is highly available to serve requests, with the ability to tune its level of availability—nearing, but never quite reaching, strong consistency. If you want strong consistency, you'll need to create a special SC bucket type, which we'll see in a later chapter.

Not Quite C

Strictly speaking, altering R and W values actually creates a tunable availability/latency tradeoff, rather than availability/consistency. Making Riak run faster by keeping R and W values low will increase the likelihood of temporarily inconsistent results (higher availability). Setting those values higher will improving the odds of consistent responses (never quite reaching strong consistency), but will slow down those responses and increase the likelihood that Riak will fail to respond (in the event of a partition).

Riak allows you to choose how many nodes you want to replicate an object to, and how many nodes must be written to or read from per request. These values are settings labeled `n_val` (the number of nodes to replicate to), `r` (the number of nodes read from before returning), and `w` (the number of nodes written to before considered successful).

A thought experiment may help clarify things.

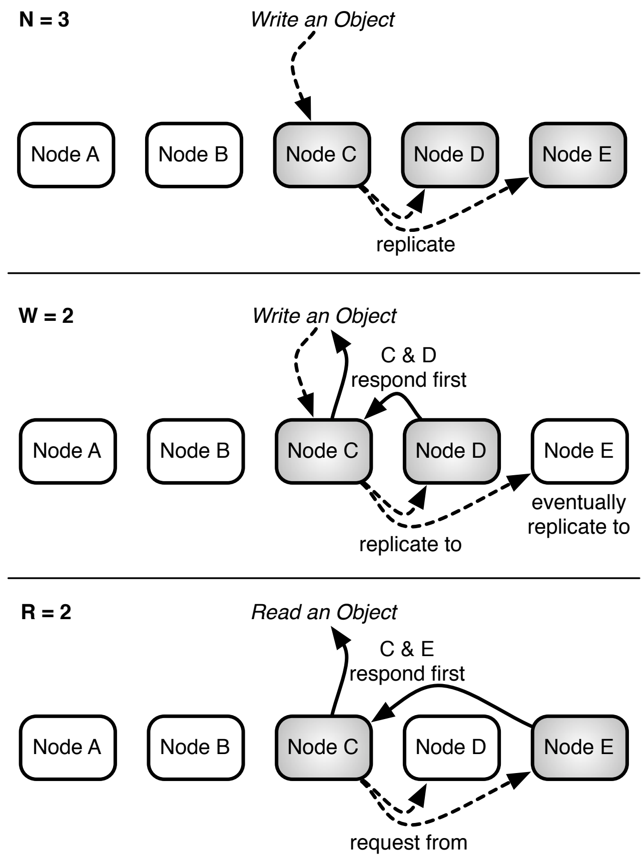


Figure 2.6: NRW

N

With our 5 node cluster, having an `n_val=3` means values will eventually replicate to 3 nodes, as we've discussed above. This is the *N value*. You can set other values (R,W) to equal the `n_val` number with the shorthand `all`.

W

But you may not wish to wait for all nodes to be written to before returning. You can choose to wait for all 3 to finish writing (`w=3` or `w=all`), which means my values are more likely to be consistent. Or you could choose to wait for only 1 complete write (`w=1`), and allow the remaining 2 nodes to write asynchronously, which returns a response quicker but increases the odds of reading an inconsistent value in the short term. This is the *W value*.

In other words, setting `w=all` would help ensure your system was more likely to be consistent, at the expense of waiting longer, with a chance that your write would fail if fewer than 3 nodes were available (meaning, over half of your total servers are down).

A failed write, however, is not necessarily a true failure. The client will receive an error message, but the write will typically still have succeeded on some number of nodes smaller than the *W* value, and will typically eventually be propagated to all of the nodes that should have it.

R

Reading involves similar tradeoffs. To ensure you have the most recent value, you can read from all 3 nodes containing objects (`r=all`). Even if only 1 of 3 nodes has the most recent value, we can compare all nodes against each other and choose the latest one, thus ensuring some consistency. Remember when I mentioned that RDBMS databases were *write consistent*? This is close to *read consistency*. Just like `w=all`, however, the read will fail unless 3 nodes are available to be read. Finally, if you only want to quickly read any value, `r=1` has low latency, and is likely consistent if `w=all`.

In general terms, the N/R/W values are Riak's way of allowing you to trade lower consistency for more availability.

Logical Clock

If you've followed thus far, I only have one more conceptual wrench to throw at you. I wrote earlier that with `r=all`, we can "compare all nodes against each other and choose the latest one." But how do we know which is the latest value? This is where logical clocks like *vector clocks* (aka *vclocks*) come into play.

DVV

Since Riak 2.0, some internal values have been migrated over to an alternative logical timestamp called Dot Version Vectors. How they operate isn't really germain to this short lesson. What's important is not so much how they're implemented, but the basic idea of a logical clock that we're talking about in the section. You can read more about DVVs (or any Riak concept) on the Basho docs website.

Vector clocks measure a sequence of events, just like a normal clock. But since we can't reasonably keep the clocks on dozens, or hundreds, or thousands of servers in sync (without really exotic hardware, like geosynchronized atomic clocks, or quantum entanglement), we instead keep a running history of updates, and look for logical, rather than temporal, causality.

Let's use our `favorite` example again, but this time we have 3 people trying to come to a consensus on their favorite food: Aaron, Britney, and Carrie. These people are called *actors*, ie. the things responsible for the updates. We'll track the value each actor has chosen along with the relevant vector clock.

(To illustrate vector clocks in action, we're cheating a bit. Riak doesn't track vector clocks via the client that initiated the request, but rather, via the server that coordinates the write request; nonetheless, the concept is the same. We'll cheat further by disregarding the timestamp that is stored with vector clocks.)

When Aaron sets the `favorite` object to `pizza`, a vector clock could contain his name and the number of updates he's performed.

```
bucket: food
key:    favorite

vclock: {Aaron: 1}
value:  pizza
```

Britney now comes along, and reads `favorite`, but decides to update `pizza` to `cold pizza`. When using vclocks, she must provide the vclock returned from the request she wants to update. This is how Riak can help ensure you're updating a previous value, and not merely overwriting with your own.

```
bucket: food
key:    favorite

vclock: {Aaron: 1, Britney: 1}
value:  cold pizza
```

At the same time as Britney, Carrie decides that pizza was a terrible choice, and tried to change the value to `lasagna`.

```
bucket: food
key:    favorite

vclock: {Aaron: 1, Carrie: 1}
value:  lasagna
```

This presents a problem, because there are now two vector clocks in play that diverge from `{Aaron: 1}`. By default, Riak will store both values.

Later in the day Britney checks again, but this time she gets the two conflicting values (aka *siblings*, which we'll discuss in more detail in the next chapter), with two vclocks.

```
bucket: food
key:    favorite

vclock: {Aaron: 1, Britney: 1}
value:  cold pizza
---
vclock: {Aaron: 1, Carrie: 1}
value:  lasagna
```

It's clear that a decision must be made. Perhaps Britney knows that Aaron's original request was for `pizza`, and thus two people generally agreed on `pizza`, so she resolves the conflict choosing that and providing a new vclock.

```
bucket: food
key:    favorite

vclock: {Aaron: 1, Carrie: 1, Britney: 2}
value:  pizza
```

Now we are back to the simple case, where requesting the value of `favorite` will just return the agreed upon `pizza`.

If you're a programmer, you may notice that this is not unlike a version control system, like **git**, where conflicting branches may require manual merging into one.

Datatypes

New in Riak 2.0 is the concept of datatypes. In the preceding logical clock example, we were responsible for resolving the conflicting values. This is because in the normal case, Riak has no idea what object's you're giving it. That is to say, Riak values are *opaque*. This is actually a powerful construct, since it allows you to store any type of value you want, from plain text, to semi-structured data like XML or JSON, to binary objects like images.

When you decide to use datatypes, you've given Riak some information about the type of object you want to store. With this information, Riak can figure out how to resolve conflicts automatically for you, based on some pre-defined behavior.

Let's try another example. Let's imagine a shopping cart in an online retailer. You can imagine a shopping cart like a set of items. So each key in our cart contains a *set* of values.

Let's say you log into the retailer's website on your laptop with your username *ponies4evr*, and choose the Season 2 DVD of *My Little Pony: Friendship is Magic*. This time, the logical clock will act more like Riak's, where the node that coordinates the request will be the actor.

```
type:   set
bucket: cart
key:    ponies4evr

vclock: {Node_A: 1}
value:  ["MYPFIM-S2-DVD"]
```

Once the DVD was added to the cart bucket, your laptop runs out of batteries. So you take out your trusty smartphone, and log into the retailer's mobile app. You decide to also add the *Bloodsport III* DVD. Little did you know, a temporary network partition caused your write to redirect to another node. This partition had no knowledge of of your other purchase.

```
type:   set
bucket: cart
key:    ponies4evr

vclock: {Node_B: 1}
value:  ["BS-III-DVD"]
```

Happily, the network hiccup was temporary, and thus the cluster heals itself. Undr normal circumstances, since the logical clocks did not decend from one another, you'd end up with siblings like this:

```
type:   set
bucket: cart
```

```
key:    ponies4evr

vclock: {Node_A: 1}
value:  ["MYPFIM-S2-DVD"]
---
vclock: {Node_B: 1}
value:  ["BS-III-DVD"]
```

But since the bucket was designed to hold a *set*, Riak knows how to automatically resolve this conflict. In the case of conflicting sets, it performs a set union. So when you go to checkout of the cart, the system returns this instead:

```
type:   set
bucket: cart
key:    ponies4evr

vclock: [{Node_A: 1}, {Node_B: 1}]
value:  ["MYPFIM-S2-DVD", "BS-III-DVD"]
```

Datatypes will never return conflicts. This is an important claim to make, because as a developer, you get all of the benefits of dealing with a simple value, with all of the benefits of a distributed, available system. You don't have to think about handling conflicts. It would be like a version control system where (*git*, *svn*, etc) where you never had to merge code—the VCS simply *knew* what you wanted.

How this all works is beyond the scope of this document. Under the covers it's implemented by something called CRDTs (Conflict-free Replicated Data Types). What's important to note is that Riak supports four datatypes: *map*, *set*, *counter*, *flag* (a boolean value). Best of all, maps can nest arbitrarily, so you can create a map whose values are sets, counters, or even other maps. It also supports plain string values called *register*s.

We'll see how to use datatypes in the next chapter.

Riak and ACID

Distributed Relational is Not Exempt

So why don't we just distribute a standard relational database? MySQL has the ability to cluster, and it's ACID (Atomic, *Consistent*, *Isolated*, *Durable*), right? Yes and no.

A single node in the cluster is ACID, but the entire cluster is not without a loss of availability and (often worse) increased latency. When you write to a primary node, and a secondary node is replicated to, a network partition can occur. To remain available, the secondary will not be in sync (eventually consistent). Have you ever loaded from a backup on database failure, but the dataset was incomplete by a few hours? Same idea.

Or, the entire transaction can fail, making the whole cluster unavailable. Even ACID databases cannot escape the scourge of CAP.

Unlike single node databases like Neo4j or PostgreSQL, Riak does not support *ACID* transactions. Locking across multiple servers would can write availability, and equally concerning, increase latency.

While ACID transactions promise *Atomicity*, *Consistency*, *Isolation*, and *Durability*—Riak and other NoSQL databases follow *BASE*, or *Basically Available*, *Soft state*, *Eventually consistent*.

The BASE acronym was meant as shorthand for the goals of non-ACID-transactional databases like Riak. It is an acceptance that distribution is never perfect (basically available), all data is in flux (soft state), and that strong consistency is untenable (eventually consistent) if you want high availability.

Look closely at promises of distributed transactions—it's often couched in some diminishing adjective or caveat like *row transactions*, or *per node transactions*, which basically mean *not transactional* in terms you would normally use to define it. I'm not claiming it's impossible, but certainly worth due consideration.

As your server count grows—especially as you introduce multiple datacenters—the odds of partitions and node failures drastically increase. My best advice is to design for it upfront.

Wrapup

Riak is designed to bestow a range of real-world benefits, but equally, to handle the fallout of wielding such power. Consistent hashing and vnodes are an elegant solution to horizontally scaling across servers. N/R/W allows you to dance with the CAP theorem by fine-tuning against its constraints. And vector clocks allow another step closer to consistency by allowing you to manage conflicts that will occur at high load.

We'll cover other technical concepts as needed, including the gossip protocol, hinted handoff, and read-repair.

Next we'll review Riak from the user (developer) perspective. We'll check out lookups, take advantage of write hooks, and examine alternative query options like secondary indexing, search, and MapReduce.

Chapter 3

Developers

A Note on "Node"

It's worth mentioning that I use the word "node" a lot. Realistically, this means a physical/virtual server, but really, the workhorses of Riak are vnodes.

When you write to multiple vnodes, Riak will attempt to spread values to as many physical servers as possible. However, this isn't guaranteed (for example, if you have only 2 physical servers with the default `n_val` of 3, some data will be copied to the same server twice). You're safe conceptualizing nodes as Riak instances, and it's simpler than qualifying "vnode" all the time. If something applies specifically to a vnode, I'll mention it.

We're going to hold off on the details of installing Riak at the moment. If you'd like to follow along, it's easy enough to get started by following the install documentation on the website (http://docs.basho.com). If not, this is a perfect section to read while you sit on a train without an Internet connection.

Developing with a Riak database is quite easy to do, once you understand some of the finer points. It is a key/value store, in the technical sense (you associate values with keys, and retrieve them using the same keys) but it offers so much more. You can embed write hooks to fire before or after a write, or index data for quick retrieval. Riak has SOLR search, and lets you run MapReduce functions to extract and aggregate data across a huge cluster in relatively short timespans. We'll show some configurable bucket-specific settings.

Lookup

Supported Languages

Riak 2.0 has official drivers for the following languages: Erlang, Java, Python, Ruby.

Including community-supplied drivers, supported languages are even more numerous: C/C++, PHP, Clojure, Common Lisp, Dart, Go, Groovy, Haskell, JavaScript (jQuery and NodeJS), Lisp Flavored Erlang, .NET, Perl, PHP, Play, Racket, Scala, Smalltalk.

Dozens of other project-specific addons can be found in the Basho docs.

Since Riak is a KV database, the most basic commands are setting and getting values. We'll use the HTTP interface, via curl, but we could just as easily use Erlang, Ruby, Java, or any other supported language.

The basic structure of a Riak request is setting a value, reading it, and maybe eventually deleting it. The actions are related to HTTP methods (PUT, GET, POST, DELETE).

```
PUT    /types/<type>/buckets/<bucket>/keys/<key>
GET    /types/<type>/buckets/<bucket>/keys/<key>
DELETE /types/<type>/buckets/<bucket>/keys/<key>
```

For the examples in this chapter, let's call an environment variable `$RIAK` that points to our access node's URL.

```
export RIAK=http://localhost:8098
```

PUT

The simplest write command in Riak is putting a value. It requires a key, value, and a bucket. In curl, all HTTP methods are prefixed with `-X`. Putting the value `pizza` into the key `favorite` under the `food` bucket and `items` bucket type is done like this:

```
curl -XPUT "$RIAK/types/items/buckets/food/keys/favorite" \
  -H "Content-Type:text/plain" \
  -d "pizza"
```

I threw a few curveballs in there. The `-d` flag denotes the next string will be the value. We've kept things simple with the string `pizza`, declaring it as text with the proceeding line `-H 'Content-Type:text/plain'`. This defines the HTTP MIME type of this value as plain text. We could have set any value at all, be it XML or JSON—even an image or a video. Riak does not care at all what data is uploaded, so long as the object size doesn't get much larger than 4MB (a soft limit but one that it is unwise to exceed).

GET

The next command reads the value `pizza` under the type/bucket/key `items/food/favorite`.

```
curl -XGET "$RIAK/types/items/buckets/food/keys/favorite"
pizza
```

This is the simplest form of read, responding with only the value. Riak contains much more information, which you can access if you read the entire response, including the HTTP header.

In `curl` you can access a full response by way of the `-i` flag. Let's perform the above query again, adding that flag (`-XGET` is the default curl method, so we can leave it off).

```
curl -i "$RIAK/types/items/buckets/food/keys/favorite"
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fgaUHjmdwZTImMfKcN3h1Um+LAA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted...
Last-Modified: Wed, 10 Oct 2012 18:56:23 GMT
ETag: "1yHn7L0XMEoMVXRGp4gOom"
Date: Thu, 11 Oct 2012 23:57:29 GMT
Content-Type: text/plain
Content-Length: 5

pizza
```

The anatomy of HTTP is a bit beyond this little book, but let's look at a few parts worth noting.

Status Codes The first line gives the HTTP version 1.1 response code `200 OK`. You may be familiar with the common website code `404 Not Found`. There are many kinds of HTTP status codes, and the Riak HTTP interface stays true to their intent: **1xx Informational**, **2xx Success**, **3xx Further Action**, **4xx Client Error**, **5xx Server Error**

Different actions can return different response/error codes. Complete lists can be found in the official API docs.

Timings A block of headers represents different timings for the object or the request.

- **Last-Modified** - The last time this object was modified (created or updated).
- **ETag** - An *entity tag* which can be used for cache validation by a client.
- **Date** - The time of the request.
- **X-Riak-Vclock** - A logical clock which we'll cover in more detail later.

Content These describe the HTTP body of the message (in Riak's terms, the *value*).

- **Content-Type** - The type of value, such as `text/xml`.
- **Content-Length** - The length, in bytes, of the message body.

Some other headers like `Link` will be covered later in this chapter.

POST

Similar to PUT, POST will save a value. But with POST a key is optional. All it requires is a bucket name (and should include a type), and it will generate a key for you.

Let's add a JSON value to represent a person under the `json/people` type/bucket. The response header is where a POST will return the key it generated for you.

```
curl -i -XPOST "$RIAK/types/json/buckets/people/keys" \
  -H "Content-Type:application/json" \
  -d '{"name":"aaron"}'
HTTP/1.1 201 Created
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.2 (someone had painted...
Location: /riak/people/DNQGJY0KtcHMirkidasA066yj5V
Date: Wed, 10 Oct 2012 17:55:22 GMT
Content-Type: application/json
Content-Length: 0
```

You can extract this key from the `Location` value. Other than not being pretty, this key is treated the same as if you defined your own key via PUT.

Body You may note that no body was returned with the response. For any kind of write, you can add the `returnbody=true` parameter to force a value to return, along with value-related headers like `X-Riak-Vclock` and `ETag`.

```
curl -i -XPOST "$RIAK/types/json/buckets/people/keys?returnbody=true" \
  -H "Content-Type:application/json" \
  -d '{"name":"billy"}'
HTTP/1.1 201 Created
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fgaUHjmdwZTImMfKkD3z10m+LAA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted...
Location: /riak/people/DnetI8GHiBK2yBFOEcj1EhHprss
Last-Modified: Tue, 23 Oct 2012 04:30:35 GMT
ETag: "7DsE7SEqAtY12d8T1HMkWZ"
Date: Tue, 23 Oct 2012 04:30:35 GMT
Content-Type: application/json
Content-Length: 16

{"name":"billy"}
```

This is true for PUTs and POSTs.

DELETE

The final basic operation is deleting keys, which is similar to getting a value, but sending the DELETE method to the `type/bucket/key`.

```
curl -XDELETE "$RIAK/types/json/buckets/people/keys/DNQGJY0KtcHMirkidasA066yj5V"
```

A deleted object in Riak is internally marked as deleted, by writing a marker known as a *tombstone*. Unless configured otherwise, another process called a *reaper* will later finish deleting the marked objects.

This detail isn't normally important, except to understand two things:

1. In Riak, a *delete* is actually a *read* and a *write*, and should be considered as such when calculating read/write ratios.

2. Checking for the existence of a key is not enough to know if an object exists. You might be reading a key after it has been deleted, so you should check for tombstone metadata.

Lists

Riak provides two kinds of lists. The first lists all *buckets* in your cluster, while the second lists all *keys* under a specific bucket. Both of these actions are called in the same way, and come in two varieties.

The following will give us all of our buckets as a JSON object.

```
curl "$RIAK/types/default/buckets?buckets=true"

{"buckets":["food"]}
```

And this will give us all of our keys under the `food` bucket.

```
curl "$RIAK/types/default/buckets/food/keys?keys=true"
{
  ...
  "keys": [
    "favorite"
  ]
}
```

If we had very many keys, clearly this might take a while. So Riak also provides the ability to stream your list of keys. `keys=stream` will keep the connection open, returning results in chunks of arrays. When it has exhausted its list, it will close the connection. You can see the details through curl in verbose (`-v`) mode (much of that response has been stripped out below).

```
curl -v "$RIAK/types/default/buckets/food/keys?keys=stream"
...

* Connection #0 to host localhost left intact
...
{"keys":["favorite"]}
{"keys":[]}
* Closing connection #0
```

You should note that list actions should *not* be used in production (they're really expensive operations). But they are useful for development, investigations, or for running occasional analytics at off-peak hours.

Conditional requests

It is possible to use conditional requests with Riak, but these are fragile due to the nature of its availability/eventual consistency model.

GET

When retrieving values from Riak via HTTP, a last-modified timestamp and an ETag are included. These may be used for future `GET` requests; if the value has not changed, a `304 Not Modified` status will be returned.

For example, let's assume you receive the following headers.

```
Last-Modified: Thu, 17 Jul 2014 21:01:16 GMT
ETag: "3VhRP0vnXbk5NjZllr0dDE"
```

Note that the quotes are part of the ETag.

If the ETag is used via the `If-None-Match` header in the next request:

```
curl -i "$RIAK/types/default/buckets/food/keys/dinner" \
  -H 'If-None-Match: "3VhRP0vnXbk5NjZllr0dDE"'
HTTP/1.1 304 Not Modified
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.5 (jokes are better explained)
ETag: "3VhRP0vnXbk5NjZllr0dDE"
Date: Mon, 28 Jul 2014 19:48:13 GMT
```

Similarly, the last-modified timestamp may be used with `If-Modified-Since`:

```
curl -i "$RIAK/types/default/buckets/food/keys/dinner" \
  -H 'If-Modified-Since: Thu, 17 Jul 2014 21:01:16 GMT'
HTTP/1.1 304 Not Modified
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.5 (jokes are better explained)
ETag: "3VhRP0vnXbk5NjZllr0dDE"
Date: Mon, 28 Jul 2014 19:51:39 GMT
```

PUT & DELETE

When adding, updating, or removing content, the HTTP headers `If-None-Match`, `If-Match`, `If-Modified-Since`, and `If-Unmodified-Since` can be used to specify ETags and timestamps.

If the specified condition cannot be met, a `412 Precondition Failed` status will be the result.

Bucket Types/Buckets

Although we've been using bucket types and buckets as namespaces up to now, they are capable of more.

Different use-cases will dictate whether a bucket is heavily written to, or largely read from. You may use one bucket to store logs, one bucket could store session data, while another may store shopping cart data. Sometimes low latency is important, while other times it's high durability. And sometimes we just want buckets to react differently when a write occurs.

Quorum

The basis of Riak's availability and tolerance is that it can read from, or write to, multiple nodes. Riak allows you to adjust these N/R/W values (which we covered under Concepts) on a per-bucket basis.

N/R/W

N is the number of total nodes that a value should be replicated to, defaulting to 3. But we can set this `n_val` to less than the total number of nodes.

Any bucket property, including `n_val`, can be set by sending a `props` value as a JSON object to the bucket URL. Let's set the `n_val` to 5 nodes, meaning that objects written to `cart` will be replicated to 5 nodes.

```
curl -i -XPUT "$RIAK/types/default/buckets/cart/props" \
  -H "Content-Type: application/json" \
  -d '{"props":{"n_val":5}}'
```

You can take a peek at the bucket's properties by issuing a GET to the bucket.

Note: Riak returns unformatted JSON. If you have a command-line tool like jsonpp (or json_pp) installed, you can pipe the output there for easier reading. The results below are a subset of all the `props` *values.*

```
curl "$RIAK/types/default/buckets/cart/props" | jsonpp
{
  "props": {
    ...
    "dw": "quorum",
    "n_val": 5,
    "name": "cart",
    "postcommit": [],
    "pr": 0,
    "precommit": [],
    "pw": 0,
    "r": "quorum",
    "rw": "quorum",
    "w": "quorum",
    ...
  }
}
```

As you can see, `n_val` is 5. That's expected. But you may also have noticed that the cart `props` returned both `r` and `w` as `quorum`, rather than a number. So what is a *quorum*?

Symbolic Values A *quorum* is one more than half of all the total replicated nodes (`floor(N/2) + 1`). This figure is important, since if more than half of all nodes are written to, and more than half of all nodes are read from, then you will get the most recent value (under normal circumstances).

Here's an example with the above `n_val` of 5 ({A,B,C,D,E}). Your `w` is a quorum (which is `3`, or `floor(5/2)+1`), so a PUT may respond successfully after writing to {A,B,C} ({D,E} will eventually be replicated to). Immediately after, a read quorum may GET values from {C,D,E}. Even if D and E have older values, you have pulled a value from node C, meaning you will receive the most recent value.

What's important is that your reads and writes *overlap*. As long as `r+w > n`, in the absence of *sloppy quorum* (below), you'll be able to get the newest values. In other words, you'll have a reasonable level of consistency.

A `quorum` is an excellent default, since you're reading and writing from a balance of nodes. But if you have specific requirements, like a log that is often written to, but rarely read, you might find it make more sense to wait for a successful write from a single node, but read from all of them. This affords you an overlap

```
curl -i -XPUT "$RIAK/types/default/buckets/logs/props" \
  -H "Content-Type: application/json" \
  -d '{"props":{"w":"one","r":"all"}}'
```

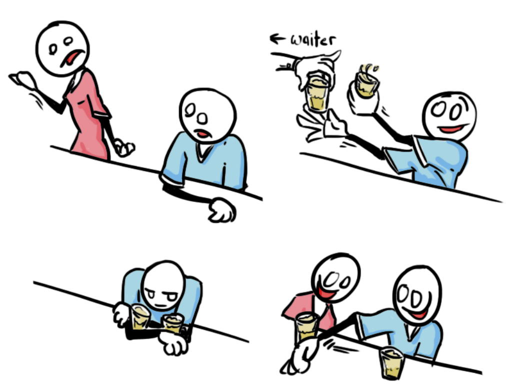
- `all` - All replicas must reply, which is the same as setting `r` or `w` equal to `n_val`
- `one` - Setting `r` or `w` equal to 1
- `quorum` - A majority of the replicas must respond, that is, "half plus one".

Sloppy Quorum

In a perfect world, a strict quorum would be sufficient for most write requests. However, at any moment a node could go down, or the network could partition, or squirrels get caught in the tubes, triggering the unavailability of a required nodes. This is known as a strict quorum. Riak defaults to what's known as a *sloppy quorum*, meaning that if any primary (expected) node is unavailable, the next available node in the ring will accept requests.

Think about it like this. Say you're out drinking with your friend. You order 2 drinks (W=2), but before they arrive, she leaves temporarily. If you were a strict quorum, you could merely refuse both drinks, since the required people (N=2) are unavailable. But you'd rather be a sloppy drunk... erm, I mean sloppy *quorum*. Rather than deny the drink, you take both, one accepted *on her behalf* (you also get to pay).

When she returns, you slide her drink over. This is known as *hinted handoff*, which we'll look at again in the next chapter. For now it's sufficient to note that there's a difference between the default sloppy quorum (W), and requiring a strict quorum of primary nodes (PW).



More than R's and W's Some other values you may have noticed in the bucket's `props` object are `pw`, `pr`, and `dw`.

`pr` and `pw` ensure that many *primary* nodes are available before a read or write. Riak will read or write from backup nodes if one is unavailable, because of network partition or some other server outage. This `p` prefix will ensure that only the primary nodes are used, *primary* meaning the vnode which matches the bucket plus N successive vnodes.

(We mentioned above that `r+w > n` provides a reasonable level of consistency, violated when sloppy quorums are involved. `pr+pw > n` allows for a much stronger assertion of consistency, although there are always scenarios involving conflicting writes or significant disk failures where that too may not be enough.)

Finally `dw` represents the minimal *durable* writes necessary for success. For a normal `w` write to count a write as successful, a vnode need only promise a write has started, with no guarantee that write has been written to disk, aka, is durable. The `dw` setting means the backend service (for example Bitcask) has agreed to write the value. Although a high `dw` value is slower than a high `w` value, there are cases where this extra enforcement is good to have, such as dealing with financial data.

Per Request It's worth noting that these values (except for `n_val`) can be overridden *per request*.

Consider a scenario in which you have data that you find very important (say, credit card checkout), and want to help ensure it will be written to every relevant node's disk before success. You could add `?dw=all` to the end of your write.

```
curl -i -XPUT "$RIAK/types/default/buckets/cart/keys/cart1?dw=all" \
  -H "Content-Type: application/json" \
  -d '{"paid":true}'
```

If any of the nodes currently responsible for the data cannot complete the request (i.e., hand off the data to the storage backend), the client will receive a failure message. This doesn't mean that the write failed, necessarily: if two of three primary vnodes successfully wrote the value, it should be available for future requests. Thus trading availability for consistency by forcing a high `dw` or `pw` value can result in unexpected behavior.

Hooks

Another utility of buckets are their ability to enforce behaviors on writes by way of hooks. You can attach functions to run either before, or after, a value is committed to a bucket.

Precommit hooks are functions that run before a write is called. A precommit hook has the ability to cancel a write altogether if the incoming data is considered bad in some way. A simple precommit hook is to check if a value exists at all.

I put my custom Erlang code files under the riak installation `./custom/my_validators.erl`.

```
-module(my_validators).
-export([value_exists/1]).

%% Object size must be greater than 0 bytes
value_exists(RiakObject) ->
  Value = riak_object:get_value(RiakObject),
  case erlang:byte_size(Value) of
    0 -> {fail, "A value sized greater than 0 is required"};
    _ -> RiakObject
  end.
```

Then compile the file.

```
erlc my_validators.erl
```

Install the file by informing the Riak installation of your new code via `app.config` (restart Riak).

```
{riak_kv,
  ...
  {add_paths, ["./custom"]}
}
```

Then you need to do set the Erlang module (`my_validators`) and function (`value_exists`) as a JSON value to the bucket's precommit array `{"mod":"my_validators","fun":"value_exists"}`.

```
curl -i -XPUT "$RIAK/types/default/buckets/cart/props" \
  -H "Content-Type:application/json" \
  -d '{"props":{"precommit":[{"mod":"my_validators","fun":"value_exists"}]}}'
```

If you try and post to the `cart` bucket without a value, you should expect a failure.

```
curl -XPOST "$RIAK/types/default/buckets/cart/keys" \
  -H "Content-Type:application/json"
A value sized greater than 0 is required
```

You can also write precommit functions in JavaScript, though Erlang code will execute faster.

Post-commits are similar in form and function, albeit executed after the write has been performed. Key differences:

- The only language supported is Erlang.
- The function's return value is ignored, thus it cannot cause a failure message to be sent to the client.

Datatypes

A new feature in Riak 2.0 are datatypes. Rather than the opaque values of days past, these new additions allow a user to define the type of values that are accepted under a given bucket type. In addition to the benefits listed in the previous chapter of automatic conflict resolution, you also interact with datatypes in a different way.

CRDT

In the previous chapter I said that Riak datatypes are implemented as CRDTs. The definition of CRDT given was Conflict-free Replicated Data Types. This is only partially correct. In fact, there are two varients of CRDTs, namely, describing how they attempt to keep the replicated datatypes Conflict-free. They are Convergent (CvRDT) and Commutative (CmRDT).

CmRDTs are datatypes that are updated with commutative operations. CvRDTs ensure that disparate states converge to a single value. This distinction is interesting in Riak, because Basho actually implements both. You interface with datatypes by commutative operations (meaning, it doesn't matter which takes place first), while any underlying divergent states will eventually converge.

In normal Riak operations, as we've seen, you put a value with a given key into a type/bucket object. If you wanted to store a map, say, as a JSON object representing a person, you would put the entire object with every field/value as an operation.

```
curl -XPOST "$RIAK/types/json/buckets/people/keys/joe" \
  -H "Content-Type:application/json"
  -d '{"name_register":"Joe", "pets_set":["cat"]}'
```

But if you wanted to add a `fish` as a pet, you'd have to replace the entire object.

```
curl -XPOST "$RIAK/types/json/buckets/people/keys/joe" \
  -H "Content-Type:application/json"
  -d '{"name_register":"Joe", "pets_set":["cat", "fish"]}'
```

As we saw in the previous chapter, this runs the risk of conflicting, thus creating a sibling.

```
{"name_register":"Joe", "pets_set":["cat"]}
{"name_register":"Joe", "pets_set":["cat", "fish"]}
```

But if we used a map, we'd instead issue only updates to create a map. So, assume that the bucket type `map` is of a map datatype (we'll see how operators can assign datatypes to bucket types in the next chapter). This command will insert a map object with two fields (`name_register` and `pets_set`).

```
curl -XPOST "$RIAK/types/map/buckets/people/keys/joe" \
  -H "Content-Type:application/json"
```

```
  -d '{
    "update": {
      "name_register": "Joe"
      "pets_set": {
        "add_all": "cat"
      }
    }
  }'
```

Next, we want to update the `pets_set` contained within `joe`'s map. Rather than set Joe's name and his pet cat, we only need to inform the object of the change. Namely, that we want to add a `fish` to his `pets_set`.

```
curl -XPOST "$RIAK/types/map/buckets/people/keys/joe" \
  -H "Content-Type:application/json"
  -d '{
    "update": {
      "pets_set": {
        "add": "fish"
      }
    }
  }'
```

This has a few benefits. Firstly, we don't need to send duplicate data. Second, it doesn't matter what order the two requests happen in, the outcome will be the same. Third, because the operations are CmRDTs, there is no possibility of a datatype returning siblings, making your client code that much easier.

As we've noted before, there are four Riak datatypes: *map*, *set*, *counter*, *flag*. The object type is set as a bucket type property. However, when populating a map, as we've seen, you must suffix the field name with the datatype that you wish to store: **map*, *set, **counter*, *flag. For plain string values, there's a special *_register datatype suffix.

You can read more about datatypes in the docs.

Entropy

Entropy is a byproduct of eventual consistency. In other words: although eventual consistency says a write will replicate to other nodes in time, there can be a bit of delay during which all nodes do not contain the same value.

That difference is *entropy*, and so Riak has created several *anti-entropy* strategies (abbreviated as *AE*). We've already talked about how an R/W quorum can deal with differing values when write/read requests overlap at least one node. Riak can repair entropy, or allow you the option to do so yourself.

Riak has two basic strategies to address conflicting writes.

Last Write Wins

The most basic, and least reliable, strategy for curing entropy is called *last write wins*. It's the simple idea that the last write based on a node's system clock will overwrite an older one. This is currently the default behavior in Riak (by virtue of the `allow_mult` property defaulting to `false`). You can also set the `last_write_wins` property to `true`, which improves performance by never retaining vector clock history.

Realistically, this exists for speed and simplicity, when you really don't care about true order of operations, or the possibility of losing data. Since it's impossible to keep server clocks truly in sync (without the proverbial geosynchronized atomic clocks), this is a best guess as to what "last" means, to the nearest millisecond.

Vector Clocks

As we saw under Concepts, *vector clocks* are Riak's way of tracking a true sequence of events of an object. Let's take a look at using vector clocks to allow for a more sophisticated conflict resolution approach than simply retaining the last-written value.

Siblings

Siblings occur when you have conflicting values, with no clear way for Riak to know which value is correct. As of Riak 2.0, as long as you use a custom (not `default`) bucket type that isn't a datatype, conflicting writes should create siblings. This is a good thing, since it ensures no data is ever lost.

In the case where you forgo a custom bucket type, Riak will try to resolve these conflicts itself if the `allow_mult` parameter is configured to `false`. You should generally always have your buckets set to retain siblings, to be resolved by the client by ensuring `allow_mult` is `true`.

```
curl -i -XPUT "$RIAK/types/default/buckets/cart/props" \
  -H "Content-Type:application/json" \
  -d '{"props":{"allow_mult":true}}'
```

Siblings arise in a couple cases.

1. A client writes a value using a stale (or missing) vector clock.
2. Two clients write at the same time with the same vector clock value.

We used the second scenario to manufacture a conflict in the previous chapter when we introduced the concept of vector clocks, and we'll do so again here.

Creating an Example Conflict

Imagine we create a shopping cart for a single refrigerator, but several people in a household are able to order food for it. Because losing orders would result in an unhappy household, Riak is using a custom bucket type `shopping` which keeps the default `allow_mult=true`.

First Casey (a vegan) places 10 orders of kale in the cart.

Casey writes `[{"item":"kale","count":10}]`.

```
curl -i -XPUT "$RIAK/types/shopping/buckets/fridge/keys/97207?returnbody=true" \
  -H "Content-Type:application/json" \
  -d '[{"item":"kale","count":10}]'
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fgaUHjmTwZTImMfKsMKK7RRfFgA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted...
Last-Modified: Thu, 01 Nov 2012 00:13:28 GMT
ETag: "2IGTrV8g1NXEfkPZ45WfAP"
Date: Thu, 01 Nov 2012 00:13:28 GMT
Content-Type: application/json
Content-Length: 28

[{"item":"kale","count":10}]
```

Note the opaque vector clock (via the `X-Riak-Vclock` header) returned by Riak. That same value will be returned with any read request issued for that key until another write occurs.

His roommate Mark, reads the order and adds milk. In order to allow Riak to track the update history properly, Mark includes the most recent vector clock with his PUT.

Mark writes `[{"item":"kale","count":10},{"item":"milk","count":1}]`.

```
curl -i -XPUT "$RIAK/types/shopping/buckets/fridge/keys/97207?returnbody=true" \
  -H "Content-Type:application/json" \
  -H "X-Riak-Vclock:a85hYGBgzGDKBVIcypz/fgaUHjmTwZTImMfKsMKK7RRfFgA="" \
  -d '[{"item":"kale","count":10},{"item":"milk","count":1}]'
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fgaUHjmTwZTIlMfKcMaK7RRfFgA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted...
Last-Modified: Thu, 01 Nov 2012 00:14:04 GMT
ETag: "62NRijQH3mRYPRybFneZaY"
Date: Thu, 01 Nov 2012 00:14:04 GMT
Content-Type: application/json
Content-Length: 54

[{"item":"kale","count":10},{"item":"milk","count":1}]
```

If you look closely, you'll notice that the vector clock changed with the second write request

- a85hYGBgzGDKBVIcypz/fgaUHjmTwZTImMfKsMKK7RRfFgA= (after the write by Casey)
- a85hYGBgzGDKBVIcypz/fgaUHjmTwZTIlMfKcMaK7RRfFgA= (after the write by Mark)

Now let's consider a third roommate, Andy, who loves almonds. Before Mark updates the shared cart with milk, Andy retrieved Casey's kale order and appends almonds. As with Mark, Andy's update includes the vector clock as it existed after Casey's original write.

Andy writes `[{"item":"kale","count":10},{"item":"almonds","count":12}]`.

```
curl -i -XPUT "$RIAK/types/shopping/buckets/fridge/keys/97207?returnbody=true" \
  -H "Content-Type:application/json" \
  -H "X-Riak-Vclock:a85hYGBgzGDKBVIcypz/fgaUHjmTwZTImMfKsMKK7RRfFgA="" \
  -d '[{"item":"kale","count":10},{"item":"almonds","count":12}]'
HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fgaUHjmTwZTInMfKoG7LdoovCwA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted...
Last-Modified: Thu, 01 Nov 2012 00:24:07 GMT
ETag: "54Nx22W9M7JUKJnLBrRehj"
Date: Thu, 01 Nov 2012 00:24:07 GMT
Content-Type: multipart/mixed; boundary=Ql3O0enxVdaMF3YlXFOdmO5bvrs
Content-Length: 491


--Ql3O0enxVdaMF3YlXFOdmO5bvrs
Content-Type: application/json
Etag: 62NRijQH3mRYPRybFneZaY
Last-Modified: Thu, 01 Nov 2012 00:14:04 GMT

[{"item":"kale","count":10},{"item":"milk","count":1}]
--Ql3O0enxVdaMF3YlXFOdmO5bvrs
Content-Type: application/json
Etag: 7kfvPXisoVBfC43IiPKYNb
Last-Modified: Thu, 01 Nov 2012 00:24:07 GMT

[{"item":"kale","count":10},{"item":"almonds","count":12}]
--Ql3O0enxVdaMF3YlXFOdmO5bvrs--
```

Whoa! What's all that?

Since there was a conflict between what Mark and Andy set the fridge value to be, Riak kept both values.

VTag

Since we're using the HTTP client, Riak returned a `300 Multiple Choices` code with a `multipart/mixed` MIME type. It's up to you to parse the results (or you can request a specific value by its Etag, also called a Vtag).

Issuing a plain get on the `shopping/fridge/97207` key will also return the vtags of all siblings.

```
curl "$RIAK/types/shopping/buckets/fridge/keys/97207"
Siblings:
62NRijQH3mRYPRybFneZaY
7kfvPXisoVBfC43IiPKYNb
```

What can you do with this tag? Namely, you request the value of a specific sibling by its `vtag`. To get the first sibling in the list (Mark's milk):

```
curl "$RIAK/types/shopping/buckets/fridge/keys/97207?vtag=62NRijQH3mRYPRybFneZaY"
[{"item":"kale","count":10},{"item":"milk","count":1}]
```

If you want to retrieve all sibling data, tell Riak that you'll accept the multipart message by adding `-H "Accept:multipart/mixed"`.

```
curl "$RIAK/types/shopping/buckets/fridge/keys/97207" \
  -H "Accept:multipart/mixed"
```

Use-Case Specific?

When siblings are created, it's up to the application to know how to deal with the conflict. In our example, do we want to accept only one of the orders? Should we remove both milk and almonds and only keep the kale? Should we calculate the cheaper of the two and keep the cheapest option? Should we merge all of the results into a single order? This is why we asked Riak not to resolve this conflict automatically… we want this flexibility.

Resolving Conflicts

When we have conflicting writes, we want to resolve them. Since that problem is typically *use-case specific*, Riak defers it to us, and our application must decide how to proceed.

For our example, let's merge the values into a single result set, taking the larger *count* if the *item* is the same. When done, write the new results back to Riak with the vclock of the multipart object, so Riak knows you're resolving the conflict, and you'll get back a new vector clock.

Successive reads will receive a single (merged) result.

```
curl -i -XPUT "$RIAK/types/shopping/buckets/fridge/keys/97207?returnbody=true" \
  -H "Content-Type:application/json" \
  -H "X-Riak-Vclock:a85hYGBgzGDKBVIcypz/fgaUHjmTwZTInMfKoG7LdoovCwA=" \
  -d '[{"item":"kale","count":10},{"item":"milk","count":1},\
      {"item":"almonds","count":12}]'
```

Last write wins vs. siblings

Your data and your business needs will dictate which approach to conflict resolution is appropriate. You don't need to choose one strategy globally; instead, feel free to take advantage of Riak's buckets to specify which data uses siblings and which blindly retains the last value written.

A quick recap of the two configuration values you'll want to set:

- `allow_mult` defaults to `false`, which means that the last write wins.
- Setting `allow_mult` to `true` instructs Riak to retain conflicting writes as siblings.
- `last_write_wins` defaults to `false`, which (perhaps counter-intuitively) still can mean that the behavior is last write wins: `allow_mult` is the key parameter for the behavioral toggle.

- Setting `last_write_wins` to true will optimize writes by assuming that previous vector clocks have no inherent value.
- Setting both `allow_mult` and `last_write_wins` to `true` is unsupported and will result in undefined behavior.

Read Repair

When a successful read happens, but not all replicas agree upon the value, this triggers a *read repair*. This means that Riak will update the replicas with the most recent value. This can happen either when an object is not found (the vnode has no copy) or a vnode contains an older value (older means that it is an ancestor of the newest vector clock). Unlike `last_write_wins` or manual conflict resolution, read repair is (obviously, I hope, by the name) triggered by a read, rather than a write.

If your nodes get out of sync (for example, if you increase the `n_val` on a bucket), you can force read repair by performing a read operation for all of that bucket's keys. They may return with `not found` the first time, but later reads will pull the newest values.

Active Anti-Entropy (AAE)

Although resolving conflicting data during get requests via read repair is sufficient for most needs, data which is never read can eventually be lost as nodes fail and are replaced.

With Riak 1.3, Basho introduced active anti-entropy to proactively identify and repair inconsistent data. This feature is also helpful for recovering data loss in the event of disk corruption or administrative error.

The overhead for this functionality is minimized by maintaining sophisticated hash trees ("Merkle trees") which make it easy to compare data sets between vnodes, but if desired the feature can be disabled.

Querying

So far we've only dealt with key-value lookups. The truth is, key-value is a pretty powerful mechanism that spans a spectrum of use-cases. However, sometimes we need to lookup data by value, rather than key. Sometimes we need to perform some calculations, or aggregations, or search.

Secondary Indexing (2i)

A *secondary index* (2i) is a data structure that lowers the cost of finding non-key values. Like many other databases, Riak has the ability to index data. However, since Riak has no real knowledge of the data it stores (they're just binary values), it uses metadata to index defined by a name pattern to be either integers or binary values.

If your installation is configured to use 2i (shown in the next chapter), simply writing a value to Riak with the header will be indexes, provided it's prefixed by `X-Riak-Index-` and suffixed by `_int` for an integer, or `_bin` for a string.

```
curl -i -XPUT $RIAK/types/shopping/buckets/people/keys/casey \
  -H "Content-Type:application/json" \
```

```
  -H "X-Riak-Index-age_int:31" \
  -H "X-Riak-Index-fridge_bin:97207" \
  -d '{"work":"rodeo clown"}'
```

Querying can be done in two forms: exact match and range. Add a couple more people and we'll see what we get: `mark` is `32`, and `andy` is `35`, they both share `97207`.

What people own `97207`? It's a quick lookup to receive the keys that have matching index values.

```
curl "$RIAK/types/shopping/buckets/people/index/fridge_bin/97207"
{"keys":["mark","casey","andy"]}
```

With those keys it's a simple lookup to get the bodies.

The other query option is an inclusive ranged match. This finds all people under the ages of `32`, by searching between `0` and `32`.

```
curl "$RIAK/types/shopping/buckets/people/index/age_int/0/32"
{"keys":["mark","casey"]}
```

That's about it. It's a basic form of 2i, with a decent array of utility.

MapReduce

MapReduce is a method of aggregating large amounts of data by separating the processing into two phases, map and reduce, that themselves are executed in parts. Map will be executed per object to convert/extract some value, then those mapped values will be reduced into some aggregate result. What do we gain from this structure? It's predicated on the idea that it's cheaper to move the algorithms to where the data lives, than to transfer massive amounts of data to a single server to run a calculation.

This method, popularized by Google, can be seen in a wide array of NoSQL databases. In Riak, you execute a MapReduce job on a single node, which then propagates to the other nodes. The results are mapped and reduced, then further reduced down to the calling node and returned.

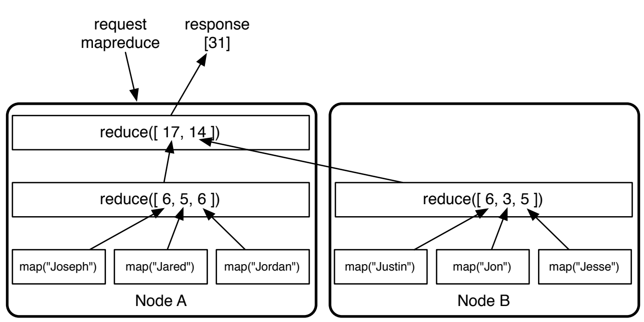


Figure 3.1: MapReduce Returning Name Char Count

Let's assume we have a bucket for log values that stores messages prefixed by either INFO or ERROR. We want to count the number of INFO logs that contain the word "cart".

```
LOGS=$RIAK/types/default/buckets/logs/keys
curl -XPOST $LOGS -d "INFO: New user added"
curl -XPOST $LOGS -d "INFO: Kale added to shopping cart"
curl -XPOST $LOGS -d "INFO: Milk added to shopping cart"
curl -XPOST $LOGS -d "ERROR: shopping cart cancelled"
```

MapReduce jobs can be either Erlang or JavaScript code. This time we'll go the easy route and write JavaScript. You execute MapReduce by posting JSON to the `/mapred` path.

```
curl -XPOST "$RIAK/mapred" \
  -H "Content-Type: application/json" \
  -d @- \
<<EOF
{
  "inputs":"logs",
  "query":[{
    "map":{
      "language":"javascript",
      "source":"function(riakObject, keydata, arg) {
        var m = riakObject.values[0].data.match(/^INFO.*cart/);
        return [(m ? m.length : 0 )];
      }"
    },
    "reduce":{
      "language":"javascript",
      "source":"function(values, arg){
        return [values.reduce(
          function(total, v){ return total + v; }, 0)
        ];
      }"
    }
  }]
}
EOF
```

The result should be `[2]`, as expected. Both map and reduce phases should always return an array. The map phase receives a single riak object, while the reduce phase received an array of values, either the result of multiple map function outputs, or of multiple reduce outputs. I probably cheated a bit by using JavaScript's `reduce` function to sum the values, but, well, welcome to the world of thinking in terms of MapReduce!

MR + 2i

Another option when using MapReduce is to combine it with secondary indexes. You can pipe the results of a 2i query into a MapReducer, simply specify the index you wish to use, and either a `key` for an index lookup, or `start` and `end` values for a ranged query.

```
    ...
    "inputs":{
       "bucket":"people",
```

```
      "index": "age_int",
      "start": 18,
      "end":   32
    },
    ...
```

MapReduce in Riak is a powerful way of pulling data out of an otherwise straight key/value store. But we have one more method of finding data in Riak.

Whatever Happened to Riak Search 1.x?

If you've used Riak before, or have some older documentation, you may wonder what the difference is between Riak Search 1.0 and 2.0.

In an attempt to make Riak Search user friendly, it was originally developed with a "Solr like" interface. Sadly, due to the complexity of building distributed search engines, it was woefully incomplete. Basho decided that, rather than attempting to maintain parity with Solr, a popular and featureful search engine in its own right, it made more sense to integrate the two.

Search 2.0

Search 2.0 is a complete, from scratch, reimagining of distributed search in Riak. It's an extension to Riak that lets you perform searches to find values in a Riak cluster. Unlike the original Riak Search, Search 2.0 leverages distributed Solr to perform the inverted indexing and management of retrieving matching values.

Before using Search 2.0, you'll have to have it installed and a bucket set up with an index (these details can be found in the next chapter).

The simplest example is a full-text search. Here we add `ryan` to the `people` table (with a default index).

```
curl -XPUT "$RIAK/type/default/buckets/people/keys/ryan" \
  -H "Content-Type:text/plain" \
  -d "Ryan Zezeski"
```

To execute a search, request `/solr/<index>/select` along with any distributed Solr parameters. Here we query for documents that contain a word starting with `zez`, request the results to be in json format (`wt=json`), only return the Riak key (`fl=_yz_rk`).

```
curl "$RIAK/solr/people/select?wt=json&omitHeader=true&fl=_yz_rk&q=zez*"
{
  "response": {
    "numFound": 1,
    "start": 0,
    "maxScore": 1.0,
    "docs": [
      {
        "_yz_rk": "ryan"
      }
```

```
    ]
  }
}
```

With the matching `_yz_rk` keys, you can retrieve the bodies with a simple Riak lookup.

Search 2.0 supports Solr 4.0, which includes filter queries, ranges, page scores, start values and rows (the last two are useful for pagination). You can also receive snippets of matching highlighted text (`hl`,`hl.fl`), which is useful for building a search engine (and something we use for search.basho.com). You can perform facet searches, stats, geolocation, bounding shapes, or any other search possible with distributed Solr.

Tagging

Another useful feature of Search 2.0 is the tagging of values. Tagging values give additional context to a Riak value. The current implementation requires all tagged values begin with `X-Riak-Meta`, and be listed under a special header named `X-Riak-Meta-yz-tags`.

```
curl -XPUT "$RIAK/types/default/buckets/people/keys/dave" \
  -H "Content-Type:text/plain" \
  -H "X-Riak-Meta-yz-tags: X-Riak-Meta-nickname_s" \
  -H "X-Riak-Meta-nickname_s:dizzy" \
  -d "Dave Smith"
```

To search by the `nickname_s` tag, just prefix the query string followed by a colon.

```
curl "$RIAK/solr/people/select?wt=json&omitHeader=true&q=nickname_s:dizzy"
{
  "response": {
    "numFound": 1,
    "start": 0,
    "maxScore": 1.4054651,
    "docs": [
      {
        "nickname_s": "dizzy",
        "id": "dave_25",
        "_yz_ed": "20121102T215100 dave m7psMIomLMu/+dtWx51Kluvvrb8=",
        "_yz_fpn": "23",
        "_yz_node": "dev1@127.0.0.1",
        "_yz_pn": "25",
        "_yz_rk": "dave",
        "_version_": 1417562617478643712
      }
    ]
  }
}
```

Notice that the `docs` returned also contain `"nickname_s":"dizzy"` as a value. All tagged values will be returned on matching results.

Datatypes

One of the more powerful combinations in Riak 2.0 are datatypes and Search. If you set both a datatype and a search index in a bucket type's properties, values you set are indexed as you'd expect. Map fields are indexed as their given types, sets are multi-field strings, counters as indexed as integers, and flags are boolean. Nested maps are also indexed, seperated by dots, and queryable in such a manner.

For example, remember Joe, from the datatype section? Let's assume that this `people` bucket is indexed. And let's also add another pet.

```
curl -XPOST "$RIAK/types/map/buckets/people/keys/joe" \
  -H "Content-Type:application/json"
  -d '{"update": {"pets_set": {"add":"dog"}}}'
```

Then let's search for `pets_set:dog`, filtering only `type/bucket/key`.

```
{
  "response": {
    "numFound": 1,
    "start": 0,
    "maxScore": 1.0,
    "docs": [
      {
        "_yz_rt": "map"
        "_yz_rb": "people"
        "_yz_rk": "joe"
      }
    ]
  }
}
```

Bravo. You've now found the object you wanted. Thanks to Solr's customizable schema, you can even store the field you want to return, if it's really that important to save a second lookup.

This provides the best of both worlds. You can update and query values without fear of conflicts, and can query Riak based on field values. It doesn't require much imagination to see that this combination effectively turns Riak into a scalable, stable, highly available, document datastore. Throw strong consistency into the mix (which we'll do in the next chapter) and you can store and query pretty much anything in Riak, in any way.

If you're wondering to yourself, "What exactly does Mongo provide, again?", well, I didn't ask it. You did. But that is a great question…

Well, moving on.

Wrapup

Riak is a distributed data store with several additions to improve upon the standard key-value lookups, like specifying replication values. Since values in Riak are opaque, many of these methods either: require custom code to extract and give meaning to values, such as *MapReduce*; or allow for header metadata to provide an added descriptive dimension to the object, such as *secondary indexes* or *search*.

Next we'll peek further under the hood, and see how to set up and manage a cluster of your own, and what you should know.

Chapter 4

Operators

In some ways, Riak is downright mundane in its role as the easiest NoSQL database to operate. Want more servers? Add them. A network cable is cut at 2am? Deal with it after a few more hours of sleep. Understanding this integral part of your application stack is still important, however, despite Riak's reliability.

We've covered the core concepts of Riak, and I've provided a taste of how to use it, but there is more to the database than that. There are details you should know if you plan on operating a Riak cluster of your own.

Clusters

Up to this point you've conceptually read about "clusters" and the "Ring" in nebulous summations. What exactly do we mean, and what are the practical implications of these details for Riak developers and operators?

A *cluster* in Riak is a managed collection of nodes that share a common Ring.

The Ring

The Ring in Riak is actually a two-fold concept.

Firstly, the Ring represents the consistent hash partitions (the partitions managed by vnodes). This partition range is treated as circular, from 0 to 2^{160}-1 back to 0 again. (If you're wondering, yes this means that we are limited to 2^{160} nodes, which is a limit of a 1.46 quindecillion, or 1.46×10^{48}, node cluster. For comparison, there are only 1.92×10^{49} silicon atoms on Earth.)

When we consider replication, the N value defines how many nodes an object is replicated to. Riak makes a best attempt at spreading that value to as many nodes as it can, so it copies to the next N adjacent nodes, starting with the primary partition and counting around the Ring, if it reaches the last partition, it loops around back to the first one.

Secondly, the Ring is also used as a shorthand for describing the state of the circular hash ring I just mentioned. This Ring (aka *Ring State*) is a data structure that gets passed around between nodes, so each knows the state of the entire cluster. Which node manages which vnodes? If a node gets a request for an object managed by other nodes, it consults the Ring and forwards the request to the proper nodes. It's a local copy of a contract that all of the nodes agree to follow.

Obviously, this contract needs to stay in sync between all of the nodes. If a node is permanently taken offline or a new one added, the other nodes need to readjust, balancing the partitions around the cluster, then updating the Ring with this new structure. This Ring state gets passed between the nodes by means of a *gossip protocol*.

Gossip

The *gossip protocol* is Riak's method of keeping all nodes current on the state of the Ring. If a node goes up or down, that information is propagated to other nodes. Periodically, nodes will also send their status to a random peer for added consistency.

Propagating changes in Ring is an asynchronous operation, and can take a couple minutes depending on Ring size.

Currently, it is not possible to change the number of vnodes of a cluster. This means that you *must have an idea of how large you want your cluster to grow in a single datacenter*. Although a basic install starts with 64 vnodes, if you plan any cluster larger than 6 or so servers you should increase vnodes to `256` or `1024`.

The number of vnodes must be a power of 2 (eg. `64`, `256`, `1024`).

Dynamic Ring resizing

A great deal of effort has been made toward being able to change the number of vnodes, so by the time you read this, it is entirely possible that Basho has released a version of Riak that allows it.

How Replication Uses the Ring

Even if you are not a programmer, it's worth taking a look at this Ring example. It's also worth remembering that partitions are managed by vnodes, and in conversation are sometimes interchanged, though I'll try to be more precise here.

Let's start with Riak configured to have 8 partitions, which are set via `ring_creation_size` in the `etc/app.config` file (we'll dig deeper into this file later).

```
%% Riak Core config
{riak_core, [
              ...
              {ring_creation_size, 8},
```

In this example, I have a total of 4 Riak nodes running on `A@10.0.1.1`, `B@10.0.1.2`, `C@10.0.1.3`, and `D@10.0.1.4`, each with two partitions (and thus vnodes)

Riak has the amazing, and dangerous, `attach` command that attaches an Erlang console to a live Riak node, with access to all of the Riak modules.

The `riak_core_ring:chash(Ring)` function extracts the total count of partitions (8), with an array of numbers representing the start of the partition, some fraction of the 2^{160} number, and the node name that represents a particular Riak server in the cluster.

```
$ bin/riak attach
(A@10.0.1.1)1> {ok,Ring} = riak_core_ring_manager:get_my_ring().
(A@10.0.1.1)2> riak_core_ring:chash(Ring).
{8,
 [{0,'A@10.0.1.1'},
  {182687704666362864775460604089535377456991567872, 'B@10.0.1.2'},
  {365375409332725729550921208179070754913983135744, 'C@10.0.1.3'},
  {548063113999088594326381812268606132370974703616, 'D@10.0.1.4'},
  {730750818665451459101842416358141509827966271488, 'A@10.0.1.1'},
  {913438523331814323877303020447676887284957839360, 'B@10.0.1.2'},
  {1096126227998177188652763624537212264741949407232, 'C@10.0.1.3'},
  {1278813932664540053428224228626747642198940975104, 'D@10.0.1.4'}]}
```

To discover which partition the bucket/key `food/favorite` object would be stored in, for example, we execute `riak_core_util:chash_key( {<<"food">>, <<"favorite">>} )` and get a wacky 160 bit Erlang number we named `DocIdx` (document index).

Just to illustrate that Erlang binary value is a real number, the next line makes it a more readable format, similar to the ring partition numbers.

```
(A@10.0.1.1)3> DocIdx =
(A@10.0.1.1)3> riak_core_util:chash_key({<<"food">>,<<"favorite">>}).
<<80,250,1,193,88,87,95,235,103,144,152,2,21,102,201,9,156,102,128,3>>

(A@10.0.1.1)4> <<I:160/integer>> = DocIdx. I.
462294600869748304160752958594990128818752487427
```

With this `DocIdx` number, we can order the partitions, starting with first number greater than `DocIdx`. The remaining partitions are in numerical order, until we reach zero, then we loop around and continue to exhaust the list.

```
(A@10.0.1.1)5> Preflist = riak_core_ring:preflist(DocIdx, Ring).
[{548063113999088594326381812268606132370974703616, 'D@10.0.1.4'},
 {730750818665451459101842416358141509827966271488, 'A@10.0.1.1'},
 {913438523331814323877303020447676887284957839360, 'B@10.0.1.2'},
 {1096126227998177188652763624537212264741949407232, 'C@10.0.1.3'},
 {1278813932664540053428224228626747642198940975104, 'D@10.0.1.4'},
 {0,'A@10.0.1.1'},
 {182687704666362864775460604089535377456991567872, 'B@10.0.1.2'},
 {365375409332725729550921208179070754913983135744, 'C@10.0.1.3'}]
```

So what does all this have to do with replication? With the above list, we simply replicate a write down the list N times. If we set N=3, then the `food/favorite` object will be written to the `D@10.0.1.4` node's partition `5480631...` (I truncated the number here), `A@10.0.1.1` partition `7307508...`, and `B@10.0.1.2` partition `9134385...`.

If something has happened to one of those nodes, like a network split (confusingly also called a partition—the "P" in "CAP"), the remaining active nodes in the list become candidates to hold the data.

So if the node coordinating the write could not reach node `A@10.0.1.1` to write to partition `7307508...`, it would then attempt to write that partition `7307508...` to `C@10.0.1.3` as a fallback (it's the next node in the list preflist after the 3 primaries).

The way that the Ring is structured allows Riak to ensure data is always written to the appropriate number of physical nodes, even in cases where one or more physical nodes are unavailable. It does this by simply trying the next available node in the preflist.

Hinted Handoff

When a node goes down, data is replicated to a backup node. This is not permanent; Riak will periodically examine whether each vnode resides on the correct physical node and hands them off to the proper node when possible.

As long as the temporary node cannot connect to the primary, it will continue to accept write and read requests on behalf of its incapacitated brethren.

Hinted handoff not only helps Riak achieve high availability, it also facilitates data migration when physical nodes are added or removed from the Ring.

Managing a Cluster

Now that we have a grasp of the general concepts of Riak, how users query it, and how Riak manages replication, it's time to build a cluster. It's so easy to do, in fact, I didn't bother discussing it for most of this book.

Install

The Riak docs have all of the information you need to install it per operating system. The general sequence is:

1. Install Erlang
2. Get Riak from a package manager (a la `apt-get` or Homebrew), or build from source (the results end up under `rel/riak`, with the binaries under `bin`).
3. Run `riak start`

Install Riak on four or five nodes—five being the recommended safe minimum for production. Fewer nodes are OK during software development and testing.

Command Line

Most Riak operations can be performed though the command line. We'll concern ourselves with two commands: `riak` and `riak-admin`.

riak

Simply typing the `riak` command will give a usage list, although not a terribly descriptive one.

```
Usage: riak {start|stop|restart|reboot|ping|console|\
            attach|chkconfig|escript|version|getpid}
```

Most of these commands are self explanatory, once you know what they mean. `start` and `stop` are simple enough. `restart` means to stop the running node and restart it inside of the same Erlang VM (virtual machine), while `reboot` will take down the Erlang VM and restart everything.

You can print the current running `version`. `ping` will return `pong` if the server is in good shape, otherwise you'll get the *just-similar-enough-to-be-annoying* response `pang` (with an *a*), or a simple `Node X not responding to pings` if it's not running at all.

`chkconfig` is useful if you want to ensure your `etc/app.config` is not broken (that is to say, it's parsable). I mentioned `attach` briefly above, when we looked into the details of the Ring—it attaches a console to the local running Riak server so you can execute Riak's Erlang code. `escript` is similar to `attach`, except you pass in script file of commands you wish to run automatically.

riak-admin

The `riak-admin` command is the meat operations, the tool you'll use most often. This is where you'll join nodes to the Ring, diagnose issues, check status, and trigger backups.

```
Usage: riak-admin { cluster | join | leave | backup | restore | test |
                    reip | js-reload | erl-reload | wait-for-service |
                    ringready | transfers | force-remove | down |
                    cluster-info | member-status | ring-status |
                    vnode-status | diag | status | transfer-limit |
                    top [-interval N] [-sort reductions|memory|msg_q]
                    [-lines N] }
```

Many of these commands are deprecated, and many don't make sense without a cluster, but a few we can look at now.

`status` outputs a list of information about this cluster. It's mostly the same information you can get from getting `/stats` via HTTP, although the coverage of information is not exact (for example, riak-admin status returns `disk`, and `/stats` returns some computed values like `gossip_received`).

```
$ riak-admin status
1-minute stats for 'A@10.0.1.1'
-------------------------------------------
vnode_gets : 0
```

```
vnode_gets_total : 2
vnode_puts : 0
vnode_puts_total : 1
vnode_index_reads : 0
vnode_index_reads_total : 0
vnode_index_writes : 0
vnode_index_writes_total : 0
vnode_index_writes_postings : 0
vnode_index_writes_postings_total : 0
vnode_index_deletes : 0
...
```

New JavaScript or Erlang files (as we did in the developers chapter) are not usable by the nodes until they are informed about them by the `js-reload` or `erl-reload` command.

`riak-admin` also provides a little `test` command, so you can perform a read/write cycle to a node, which I find useful for testing a client's ability to connect, and the node's ability to write.

Finally, `top` is an analysis command checking the Erlang details of a particular node in real time. Different processes have different process ids (Pids), use varying amounts of memory, queue up so many messages at a time (MsgQ), and so on. This is useful for advanced diagnostics, and is especially useful if you know Erlang or need help from other users, the Riak team, or Basho.

```
========================================================================================
 'B@10.0.1.2'                                                     04:25:06
 Load:  cpu         0               Memory:  total       19597    binary         97
        procs     562                        processes    4454    code         9062
        runq        0                        atom          420    ets           994

Pid                 Name or Initial Func        Time       Reds      Memory      MsgQ Current Function
----------------------------------------------------------------------------------------
<6132.154.0>        riak_core_vnode_manager      '-'       7426       90240         0 gen_server:loop/6
<6132.62.0>         timer_server                 '-'       5653        2928         0 gen_server:loop/6
<6132.61.0>         riak_sysmon_filter           '-'       4828        5864         0 gen_server:loop/6
<6132.155.0>        riak_core_capability         '-'       1425       13720         0 gen_server:loop/6
<6132.149.0>        riak_core_ring_manager       '-'       1161       88512         0 gen_server2:process_next_msg/9
<6132.156.0>        riak_core_gossip             '-'        769       34392         0 gen_server:loop/6
<6132.542.0>        mi_scheduler                 '-'         35        2848         0 gen_server:loop/6
<6132.1552.0>       inet_tcp_dist:do_accept/6    '-'         30        2744         0 dist_util:con_loop/9
<6132.1554.0>       inet_tcp_dist:do_accept/6    '-'         30        2744         0 dist_util:con_loop/9
<6132.20.0>         net_kernel                   '-'         29        4320         0 gen_server:loop/6
```

Figure 4.1: Top

Making a Cluster

With several solitary nodes running—assuming they are networked and are able to communicate to each other—launching a cluster is the simplest part.

Executing the `cluster` command will output a descriptive set of commands.

```
$ riak-admin cluster
The following commands stage changes to cluster membership. These commands
do not take effect immediately. After staging a set of changes, the staged
plan must be committed to take effect:

 join <node>                  Join node to the cluster containing <node>
 leave                        Have this node leave the cluster and shutdown
 leave <node>                 Have <node> leave the cluster and shutdown

 force-remove <node>          Remove <node> from the cluster without
                              first handing off data. Designed for
                              crashed, unrecoverable nodes
```

```
 replace <node1> <node2>       Have <node1> transfer all data to <node2>,
                               and then leave the cluster and shutdown

 force-replace <node1> <node2>  Reassign all partitions owned by <node1>
                               to <node2> without first handing off data,
                               and remove <node1> from the cluster.

Staging commands:
 plan                          Display the staged changes to the cluster
 commit                        Commit the staged changes
 clear                         Clear the staged changes
```

To create a new cluster, you must `join` another node (any will do). Taking a node out of the cluster uses `leave` or `force-remove`, while swapping out an old node for a new one uses `replace` or `force-replace`.

I should mention here that using `leave` is the nice way of taking a node out of commission. However, you don't always get that choice. If a server happens to explode (or simply smoke ominously), you don't need its approval to remove it from the cluster, but can instead mark it as `down`.

But before we worry about removing nodes, let's add some first.

```
$ riak-admin cluster join A@10.0.1.1
Success: staged join request for 'B@10.0.1.2' to 'A@10.0.1.1'
$ riak-admin cluster join A@10.0.1.1
Success: staged join request for 'C@10.0.1.3' to 'A@10.0.1.1'
```

Once all changes are staged, you must review the cluster `plan`. It will give you all of the details of the nodes that are joining the cluster, and what it will look like after each step or *transition*, including the `member-status`, and how the `transfers` plan to handoff partitions.

Below is a simple plan, but there are cases when Riak requires multiple transitions to enact all of your requested actions, such as adding and removing nodes in one stage.

```
$ riak-admin cluster plan
=============================== Staged Changes ==============
Action         Nodes(s)
-------------------------------------------------------------
join           'B@10.0.1.2'
join           'C@10.0.1.3'
-------------------------------------------------------------


NOTE: Applying these changes will result in 1 cluster transition

#############################################################
                         After cluster transition 1/1
#############################################################

================================= Membership ================
```

```
Status     Ring    Pending    Node
-------------------------------------------------------------
valid     100.0%     34.4%    'A@10.0.1.1'
valid       0.0%     32.8%    'B@10.0.1.2'
valid       0.0%     32.8%    'C@10.0.1.3'
-------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0

WARNING: Not all replicas will be on distinct nodes

Transfers resulting from cluster changes: 42
  21 transfers from 'A@10.0.1.1' to 'C@10.0.1.3'
  21 transfers from 'A@10.0.1.1' to 'B@10.0.1.2'
```

Making changes to cluster membership can be fairly resource intensive, so Riak defaults to only performing 2 transfers at a time. You can choose to alter this `transfer-limit` using `riak-admin`, but bear in mind the higher the number, the greater normal operations will be impinged.

At this point, if you find a mistake in the plan, you have the chance to `clear` it and try again. When you are ready, `commit` the cluster to enact the plan.

```
$ dev1/bin/riak-admin cluster commit
Cluster changes committed
```

Without any data, adding a node to a cluster is a quick operation. However, with large amounts of data to be transferred to a new node, it can take quite a while before the new node is ready to use.

Status Options

To check on a launching node's progress, you can run the `wait-for-service` command. It will output the status of the service and stop when it's finally up. In this example, we check the `riak_kv` service.

```
$ riak-admin wait-for-service riak_kv C@10.0.1.3
riak_kv is not up: []
riak_kv is not up: []
riak_kv is up
```

You can get a list of available services with the `services` command.

You can also see if the whole ring is ready to go with `ringready`. If the nodes do not agree on the state of the ring, it will output `FALSE`, otherwise `TRUE`.

```
$ riak-admin ringready
TRUE All nodes agree on the ring ['A@10.0.1.1','B@10.0.1.2',
                                  'C@10.0.1.3']
```

For a more complete view of the status of the nodes in the ring, you can check out `member-status`.

```
$ riak-admin member-status
================================= Membership ================
Status     Ring    Pending    Node
-------------------------------------------------------------
valid      34.4%      --      'A@10.0.1.1'
valid      32.8%      --      'B@10.0.1.2'
valid      32.8%      --      'C@10.0.1.3'
-------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
```

And for more details of any current handoffs or unreachable nodes, try `ring-status`. It also lists some information from `ringready` and `transfers`. Below I turned off the C node to show what it might look like.

```
$ riak-admin ring-status
================================== Claimant =================
Claimant:  'A@10.0.1.1'
Status:     up
Ring Ready: true

============================== Ownership Handoff ============
Owner:      dev1 at 127.0.0.1
Next Owner: dev2 at 127.0.0.1

Index: 182687704666362864775460604089535377456991567872
  Waiting on: []
  Complete:   [riak_kv_vnode,riak_pipe_vnode]
...

============================== Unreachable Nodes ============
The following nodes are unreachable: ['C@10.0.1.3']

WARNING: The cluster state will not converge until all nodes
are up. Once the above nodes come back online, convergence
will continue. If the outages are long-term or permanent, you
can either mark the nodes as down (riak-admin down NODE) or
forcibly remove the nodes from the cluster (riak-admin
force-remove NODE) to allow the remaining nodes to settle.
```

If all of the above information options about your nodes weren't enough, you can list the status of each vnode per node, via `vnode-status`. It'll show each vnode by its partition number, give any status information, and a count of each vnode's keys. Finally, you'll get to see each vnode's backend type—something I'll cover in the next section.

```
$ riak-admin vnode-status
Vnode status information
-------------------------------------------

VNode: 0
Backend: riak_kv_bitcask_backend
Status:
```

```
[{key_count,0},{status,[]}]

VNode: 913438523331814323877303020447676887284957839360
Backend: riak_kv_bitcask_backend
Status:
[{key_count,0},{status,[]}]

VNode: 1826877046663628647754606040895353774569915678720
Backend: riak_kv_bitcask_backend
Status:
[{key_count,0},{status,[]}]

VNode: 2740315569995442971631909061343030661854873518080
Backend: riak_kv_bitcask_backend
Status:
[{key_count,0},{status,[]}]

VNode: 3653754093327257295509212081790707549139831357440
Backend: riak_kv_bitcask_backend
Status:
[{key_count,0},{status,[]}]
...
```

Some commands we did not cover are either deprecated in favor of their `cluster` equivalents (`join`, `leave`, `force-remove`, `replace`, `force-replace`), or flagged for future removal `reip` (use `cluster replace`).

The last command is `diag`, which leverages Riaknostic to give you more diagnostic tools.

I know this was a lot to digest, and probably pretty dry. Walking through command line tools usually is. There are plenty of details behind many of the `riak-admin` commands, too numerous to cover in such a short book. I encourage you to toy around with them on your own installation.

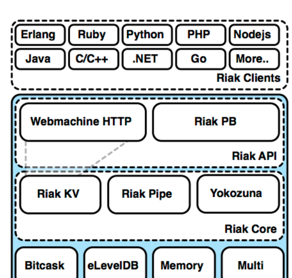
How Riak is Built

It's difficult to label Riak as a single project. It's probably more correct to think of Riak as the center of gravity for a whole system of projects. As we've covered before, Riak is built on Erlang, but that's not the whole story. It's more correct to say Riak is fundamentally Erlang, with some pluggable native C code components (like leveldb), Java (Yokozuna), and even JavaScript (for MapReduce or commit hooks).

The way Riak stacks technologies is a good thing to keep in mind, in order to make sense of how to configure it properly.

Erlang

When you fire up a Riak node, it also starts up an Erlang VM (virtual machine) to run and manage Riak's processes. These include vnodes, process messages, gossips, resource management and more. The Erlang operating system pro-



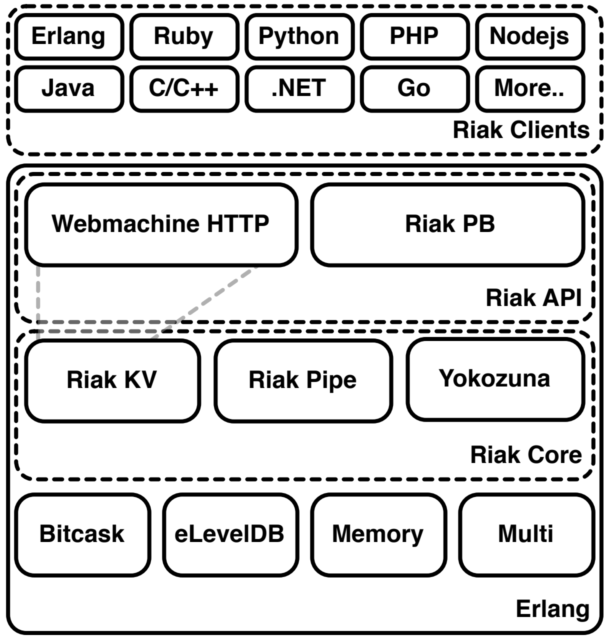


Figure 4.2: Tech Stack

cess is found as a `beam.smp` command with many, many arguments.

These arguments are configured through the `etc/vm.args` file. There are a few settings you should pay special attention to.

```
$ ps -o command | grep beam
/riak/erts-5.9.1/bin/beam.smp \
-K true \
-A 64 \
-W w -- \
-root /riak \
-progname riak -- \
-home /Users/ericredmond -- \
-boot /riak/releases/2.0.0/riak \
-embedded \
-config /riak/etc/app.config \
-pa ./lib/basho-patches \
-name A@10.0.1.1 \
-setcookie testing123 -- \
console
```

The `name` setting is the name of the current Riak node. Every node in your cluster needs a different name. It should have the IP address or dns name of the server this node runs on, and optionally a different prefix—though some people just like to name it *riak* for simplicity (eg: `riak@node15.myhost`).

The `setcookie` parameter is a setting for Erlang to perform inter-process communication (IPC) across nodes. Every node in the cluster must have the same cookie name. I recommend you change the name from `riak` to something a little less likely to accidentally conflict, like `hihohihoitsofftoworkwego`.

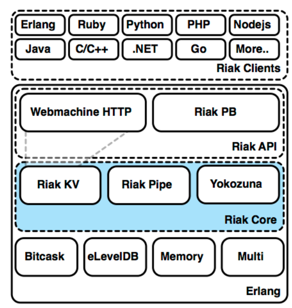
My `vm.args` starts with this:

```
## Name of the riak node
-name A@10.0.1.1

## Cookie for distributed erlang.  All nodes in the
## same cluster should use the same cookie or they
## will not be able to communicate.
-setcookie testing123
```

Continuing down the `vm.args` file are more Erlang settings, some environment variables that are set up for the process (prefixed by `-env`), followed by some optional SSL encryption settings.

riak_core

If any single component deserves the title of "Riak proper", it would be *Riak Core*. Core shares responsibility with projects built atop it for managing the partitioned keyspace, launching and supervising vnodes, preference list building, hinted handoff, and things that aren't related specifically to client interfaces, handling requests, or storage.



Riak Core, like any project, has some hard-coded values (for example, how protocol buffer messages are encoded in binary). However, many values can be modified to fit your use case. The majority of this configuration occurs under `app.config`. This file is Erlang code, so commented lines begin with a `%` character.

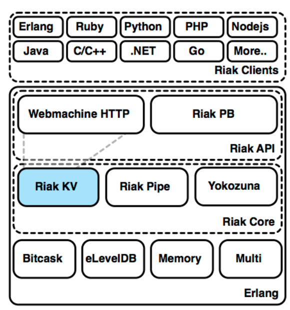
The `riak_core` configuration section allows you to change the options in this project. This handles basic settings, like files/directories where values are stored or to be written to, the number of partitions/vnodes in the cluster (`ring_creation_size`), and several port options.

```
%% Riak Core config
{riak_core, [
    %% Default location of ringstate
    {ring_state_dir, "./data/ring"},

    %% Default ring creation size.  Make sure it is a power of 2,
    %% e.g. 16, 32, 64, 128, 256, 512 etc
    %{ring_creation_size, 64},

    %% http is a list of IP addresses and TCP ports that
    %% the Riak HTTP interface will bind.
    {http, [ {"127.0.0.1", 8098 } ]},
```

```
    %% https is a list of IP addresses and TCP ports that
    %% the Riak HTTPS interface will bind.
    %{https, [{ "127.0.0.1", 8098 }]},

    %% Default cert and key locations for https can be
    %% overridden with the ssl config variable, for example:
    %{ssl, [
    %       {certfile, "./etc/cert.pem"},
    %       {keyfile, "./etc/key.pem"}
    %      ]},

    %% riak handoff_port is the TCP port that Riak uses for
    %% intra-cluster data handoff.
    {handoff_port, 8099 },

    %% To encrypt riak_core intra-cluster data handoff traffic,
    %% uncomment the following line and edit its path to an
    %% appropriate certfile and keyfile.  (This example uses a
    %% single file with both items concatenated together.)
    {handoff_ssl_options, [{certfile, "/tmp/erlserver.pem"}]},

    %% Platform-specific installation paths
    {platform_bin_dir, "./bin"},
    {platform_data_dir, "./data"},
    {platform_etc_dir, "./etc"},
    {platform_lib_dir, "./lib"},
    {platform_log_dir, "./log"}
]},
```

riak_kv

Riak KV is a key/value implementation of Riak Core. This is where the magic happens, such as handling requests and coordinating them for redundancy and read repair. It's what makes Riak a KV store rather than something else like a Cassandra-style columnar data store.

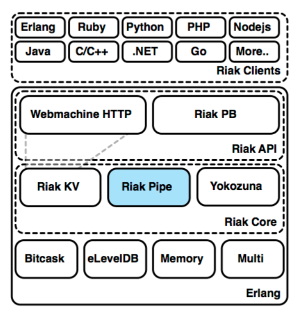


HTTP access to KV defaults to the `/riak` path as we've seen in examples throughout the book. This prefix is editable via `raw_name`. Many of the other KV settings are concerned with backward compatibility modes, backend settings, MapReduce, and JavaScript integration.

```
%% Riak KV config
{riak_kv, [
  %% raw_name is the first part of all URLS used by the
  %% Riak raw HTTP interface. See riak_web.erl and
  %% raw_http_resource.erl for details.
  {raw_name, "riak"},

  %% http_url_encoding determines how Riak treats URL
```

```
  %% encoded buckets, keys, and links over the REST API.
  %% When set to 'on'. Riak always decodes encoded values
  %% sent as URLs and Headers.
  %% Otherwise, Riak defaults to compatibility mode where
  %% links are decoded, but buckets and keys are not. The
  %% compatibility mode will be removed in a future release.
  {http_url_encoding, on},

  %% Switch to vnode-based vclocks rather than client ids.
  %% This significantly reduces the number of vclock entries.
  {vnode_vclocks, true},

  %% This option toggles compatibility of keylisting with
  %% 1.0 and earlier versions.  Once a rolling upgrade to
  %% a version > 1.0 is completed for a cluster, this
  %% should be set to true for better control of memory
  %% usage during key listing operations
  {listkeys_backpressure, true},
  ...
]},
```

riak_pipe



Riak Pipe is an input/output messaging system that forms the basis of Riak's MapReduce. This was not always the case, and MR used to be a dedicated implementation, hence some legacy options. Like the ability to alter the KV path, you can also change HTTP from `/mapred` to a custom path.

```
%% Riak KV config
{riak_kv, [
  %% mapred_name is URL used to submit map/reduce requests
  %% to Riak.
  {mapred_name, "mapred"},

  %% mapred_system indicates which version of the MapReduce
  %% system should be used: 'pipe' means riak_pipe will
  %% power MapReduce queries, while 'legacy' means that luke
  %% will be used
  {mapred_system, pipe},

  %% mapred_2i_pipe indicates whether secondary-index
  %% MapReduce inputs are queued in parallel via their
  %% own pipe ('true'), or serially via a helper process
  %% ('false' or undefined).  Set to 'false' or leave
  %% undefined during a rolling upgrade from 1.0.
  {mapred_2i_pipe, true},

  %% directory used to store a transient queue for pending
  %% map tasks
  %% Only valid when mapred_system == legacy
  %% {mapred_queue_dir, "./data/mr_queue" },
```

```
  %% Number of items the mapper will fetch in one request.
  %% Larger values can impact read/write performance for
  %% non-MapReduce requests.
  %% Only valid when mapred_system == legacy
  %% {mapper_batch_size, 5},

  %% Number of objects held in the MapReduce cache. These
  %% will be ejected when the cache runs out of room or the
  %% bucket/key pair for that entry changes
  %% Only valid when mapred_system == legacy
  %% {map_cache_size, 10000},
  ...
]}
```

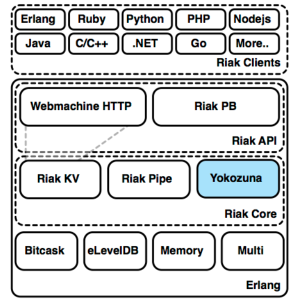
JavaScript

Riak KV's MapReduce implementation (under riak_kv, though implemented in Pipe) is the primary user of the Spidermonkey JavaScript engine—the second user is precommit hooks.

```
%% Riak KV config
{riak_kv, [
  ...
  %% Each of the following entries control how many
  %% Javascript virtual machines are available for
  %% executing map, reduce, pre- and post-commit
  %% hook functions.
  {map_js_vm_count, 8 },
  {reduce_js_vm_count, 6 },
  {hook_js_vm_count, 2 },

  %% js_max_vm_mem is the maximum amount of memory,
  %% in megabytes, allocated to the Javascript VMs.
  %% If unset, the default is 8MB.
  {js_max_vm_mem, 8},

  %% js_thread_stack is the maximum amount of thread
  %% stack, in megabyes, allocate to the Javascript VMs.
  %% If unset, the default is 16MB. NOTE: This is not
  %% the same as the C thread stack.
  {js_thread_stack, 16},

  %% js_source_dir should point to a directory containing Javascript
  %% source files which will be loaded by Riak when it initializes
  %% Javascript VMs.
  %{js_source_dir, "/tmp/js_source"},
  ...
]}
```

yokozuna

Yokozuna is the newest addition to the Riak ecosystem. It's an integration of the distributed Solr search engine into Riak, and provides some extensions for extracting, indexing, and tagging documents. The Solr server runs its own HTTP interface, and though your Riak users should never have to access it, you can choose which `solr_port` will be used.




```
%% Yokozuna Search
{yokozuna, [
  {solr_port, "8093"},
  {yz_dir, "./data/yz"}
]}
```

bitcask, eleveldb, memory, multi

Several modern databases have swappable backends, and Riak is no different in that respect. Riak currently supports three different storage engines: *Bitcask*, *eLevelDB*, and *Memory* — and one hybrid called *Multi*.

Using a backend is simply a matter of setting the `storage_backend` with one of the following values.

- `riak_kv_bitcask_backend` - The catchall Riak backend. If you don't have a compelling reason to *not* use it, this is my suggestion.
- `riak_kv_eleveldb_backend` - A Riak-friendly backend which uses Google's leveldb. This is necessary if you have too many keys to fit into memory, or wish to use 2i.
- `riak_kv_memory_backend` - A main-memory backend, with time-to-live (TTL). Meant for transient data.
- `riak_kv_multi_backend` - Any of the above backends, chosen on a per-bucket basis.

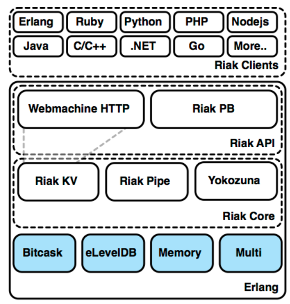
```
%% Riak KV config
{riak_kv, [
  %% Storage_backend specifies the Erlang module defining
  %% the storage mechanism that will be used on this node.
  {storage_backend, riak_kv_memory_backend}
]},
```

Then, with the exception of Multi, each memory configuration is under one of the following options.

```
%% Memory Config
{memory_backend, [
  {max_memory, 4096}, %% 4GB in megabytes
  {ttl, 86400}  %% 1 Day in seconds
]}

%% Bitcask Config
```

```
{bitcask, [
  {data_root, "./data/bitcask"},
  {open_timeout, 4}, %% Wait time to open a keydir (in seconds)
  {sync_strategy, {seconds, 60}}  %% Sync every 60 seconds
]},

%% eLevelDB Config
{eleveldb, [
  {data_root, "./data/leveldb"},
  {write_buffer_size_min, 31457280 }, %% 30 MB in bytes
  {write_buffer_size_max, 62914560}, %% 60 MB in bytes
  %% Maximum number of files open at once per partition
  {max_open_files, 20},
  %% 8MB default cache size per-partition
  {cache_size, 8388608}
]},
```

With the Multi backend, you can even choose different backends for different buckets. This can make sense, as one bucket may hold user information that you wish to index (use eleveldb), while another bucket holds volatile session information that you may prefer to simply remain resident (use memory).




```
%% Riak KV config
{riak_kv, [
  ...
  %% Storage_backend specifies the Erlang module defining
  %% the storage mechanism that will be used on this node.
  {storage_backend, riak_kv_multi_backend},

  %% Choose one of the names you defined below
  {multi_backend_default, <<"bitcask_multi">>},

  {multi_backend, [
    %% Heres where you set the individual backends
    {<<"bitcask_multi">>,  riak_kv_bitcask_backend, [
      %% bitcask configuration
      {config1, ConfigValue1},
      {config2, ConfigValue2}
    ]},
    {<<"memory_multi">>,   riak_kv_memory_backend, [
      %% memory configuration
      {max_memory, 8192}   %% 8GB
    ]}
  ]},
]},
```

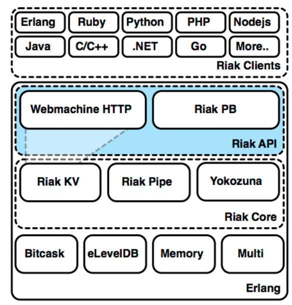
You can put the `memory_multi` configured above to the `session_data` bucket by just setting its `backend` property.

```
$ curl -XPUT http://riaknode:8098/riak/session_data \
  -H "Content-Type: application/json" \
  -d '{"props":{"backend":"memory_multi"}}'
```

riak_api

So far, all of the components we've seen have been inside the Riak house. The API is the front door. *In a perfect world*, the API would manage two implementations: HTTP and Protocol buffers (PB), an efficient binary protocol framework designed by Google.

But because they are not yet separated, only PB is configured under `riak_api`, while HTTP still remains under KV.



In any case, Riak API represents the client facing aspect of Riak. Implementations handle how data is encoded and transferred, and this project handles the services for presenting those interfaces, managing connections, providing entry points.

```
%% Riak Client APIs config
{riak_api, [
  %% pb_backlog is the maximum length to which the queue
  %% of pending connections may grow. If set, it must be
  %% an integer >= 0. By default the value is 5. If you
  %% anticipate a huge number of connections being
  %% initialised *simultaneously*, set this number higher.
  %% {pb_backlog, 64},

  %% pb_ip is the IP address that the Riak Protocol
  %% Buffers interface will bind to.  If this is undefined,
  %% the interface will not run.
  {pb_ip,   "127.0.0.1" },

  %% pb_port is the TCP port that the Riak Protocol
  %% Buffers interface will bind to
  {pb_port, 8087 }
]},
```

Other projects

Other projects add depth to Riak but aren't strictly necessary. Two of these projects are lager, for logging, and riak_sysmon, for monitoring. Both have reasonable defaults and well-documented settings.

- https://github.com/basho/lager
- https://github.com/basho/riak_sysmon

```
%% Lager Config
{lager, [
```

```
  %% What handlers to install with what arguments
  %% If you wish to disable rotation, you can either set
  %% the size to 0 and the rotation time to "", or instead
  %% specify 2-tuple that only consists of {Logfile, Level}.
  {handlers, [
    {lager_file_backend, [
      {"./log/error.log", error, 10485760, "$D0", 5},
      {"./log/console.log", info, 10485760, "$D0", 5}
    ]}
  ]},

  %% Whether to write a crash log, and where.
  %% Commented/omitted/undefined means no crash logger.
  {crash_log, "./log/crash.log"},

  ...

  %% Whether to redirect error_logger messages into lager -
  %% defaults to true
  {error_logger_redirect, true}
]},

%% riak_sysmon config
{riak_sysmon, [
  %% To disable forwarding events of a particular type, set 0
  {process_limit, 30},
  {port_limit, 2},

  %% Finding reasonable limits for a given workload is a matter
  %% of experimentation.
  {gc_ms_limit, 100},
  {heap_word_limit, 40111000},

  %% Configure the following items to 'false' to disable logging
  %% of that event type.
  {busy_port, true},
  {busy_dist_port, true}
]},
```

Backward Incompatibility

Riak is a project in evolution. And as such, it has a lot of projects that have been created, but over time are being replaced with newer versions. Obviously this baggage can be confounding if you are just learning Riak—especially as you run across deprecated configuration, or documentation.

- InnoDB - The MySQL engine once supported by Riak, but now deprecated.
- Luke - The legacy MapReduce implementation replaced by Riak Pipe.
- Search - The search implementation replaced by Yokozuna.
- Merge Index - The backend created for the legacy Riak Search.
- SASL - A logging engine improved by Lager.

Tools

Riaknostic

You may recall that we skipped the `diag` command while looking through `riak-admin`, but it's time to circle back around.

Riaknostic is a diagnostic tool for Riak, meant to run a suite of checks against an installation to discover potential problems. If it finds any, it also recommends potential resolutions.

Riaknostic exists separately from the core project but as of Riak 1.3 is included and installed with the standard database packages.

```
$ riak-admin diag --list
Available diagnostic checks:

  disk                 Data directory permissions and atime
  dumps                Find crash dumps
  memory_use           Measure memory usage
  nodes_connected      Cluster node liveness
  ring_membership      Cluster membership validity
  ring_preflists       Check ring satisfies n_val
  ring_size            Ring size valid
  search               Check whether search is enabled on all nodes
```

I'm a bit concerned that my disk might be slow, so I ran the `disk` diagnostic.

```
$ riak-admin diag disk
21:52:47.353 [notice] Data directory /riak/data/bitcask is\
not mounted with 'noatime'. Please remount its disk with the\
'noatime' flag to improve performance.
```

Riaknostic returns an analysis and suggestion for improvement. Had my disk configuration been ok, the command would have returned nothing.

Riak Control

The last tool we'll look at is the aptly named Riak Control. It's a web application for managing Riak clusters, watching, and drilling down into the details of your nodes to get a comprehensive view of the system. That's the idea, anyway. It's forever a work in progress, and it does not yet have parity with all of the command-line tools we've looked at. However, it's great for quick checkups and routing configuration changes.

Riak Control is shipped with Riak as of version 1.1, but turned off by default. You can enable it on one of your servers by editing `app.config` and restarting the node.

If you're going to turn it on in production, do so carefully: you're opening up your cluster to remote administration using a password that sadly must be stored in plain text in the configuration file.

The first step is to enable SSL and HTTPS in the `riak_core` section of `app.config`. You can just uncomment these lines, set the `https` port to a reasonable value like `8069`, and point the `certfile` and `keyfile` to your SSL certificate. If you have an intermediate authority, add the `cacertfile` too.

```
%% Riak Core config
{riak_core, [
    %% https is a list of IP addresses and TCP ports that
    %% the Riak HTTPS interface will bind.
    {https, [{ "127.0.0.1", 8069 }]},

    %% Default cert and key locations for https can be
    %% overridden with the ssl config variable, for example:
    {ssl, [
           {certfile, "./etc/cert.pem"},
           {keyfile, "./etc/key.pem"},
           {cacertfile, "./etc/cacert.pem"}
          ]},
```

Then, you'll have to `enable` Riak Control in your `app.config`, and add a user. Note that the user password is plain text. Yeah it sucks, so be careful to not open your Control web access to the rest of the world, or you risk giving away the keys to the kingdom.

```
%% riak_control config
{riak_control, [
  %% Set to false to disable the admin panel.
  {enabled, true},

  %% Authentication style used for access to the admin
  %% panel. Valid styles are 'userlist' <TODO>.
  {auth, userlist},

  %% If auth is set to 'userlist' then this is the
  %% list of usernames and passwords for access to the
  %% admin panel.
  {userlist, [{"admin", "lovesecretsexgod"}
             ]},

  %% The admin panel is broken up into multiple
  %% components, each of which is enabled or disabled
  %% by one of these settings.
  {admin, true}
]}
```

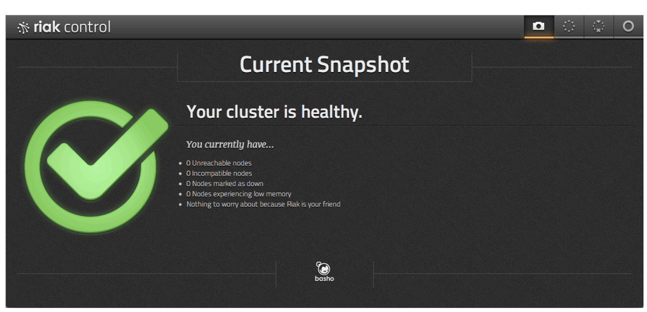


Figure 4.3: Snapshot View

With Control in place, restart your node and connect via a browser (note you're using `https`) `https://localhost:8069/admin`. After you log in using the user you set, you should see a snapshot page, which communicates the health of your cluster.

If something is wrong, you'll see a huge red "X" instead of the green check mark, along with a list of what the trouble is.

From here you can drill down into a view of the cluster's nodes, with details on memory usage, partition distribution, and other status. You can also add and configure these nodes, then view the plan and status of those changes.

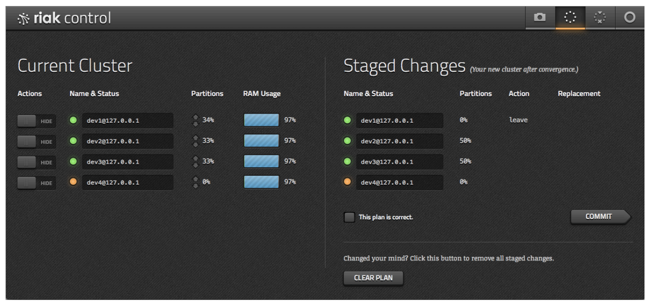


Figure 4.4: Cluster View

There is more in line for Riak Control, like performing MapReduce queries, stats views, graphs, and more coming down the pipe. It's not a universal toolkit quite yet, but it has a phenomenal start.

Once your cluster is to your liking, you can manage individual nodes, either stopping or taking them down permanently. You can also find a more detailed view of an individual node, such as what percentage of the cluster it manages, or its RAM usage.

Figure 4.5: Node Management View

Wrapup

Once you comprehend the basics of Riak, it's a simple thing to manage. If this seems like a lot to swallow, take it from a long-time relational database guy (me), Riak is a comparatively simple construct, especially when you factor in the complexity of distributed systems in general. Riak manages much of the daily tasks an operator might do themselves manually, such as sharding by keys, adding/removing nodes, rebalancing data, supporting multiple backends, and allowing growth with unbalanced nodes. And due to Riak's architecture, the best part of all is when a server goes down at night, you can sleep (do you remember what that was?), and fix it in the morning.

Chapter 5

Writing Riak Applications

Chapters 2 and 3 covered key concepts that every developer should know. In this chapter we look more closely at ways to build (and more importantly not to build) Riak applications.

How not to write a Riak application

Writing a Riak application is very much **not** like writing an application that relies on a relational database. The core ideas and vocabulary from database theory still apply, of course, but many of the decisions that inform the application layer are inverted.

Effectively all of these anti-patterns make some degree of sense when writing an application against a SQL database. None of them lend themselves to great Riak applications.

Dynamic querying

Riak's tools for finding data (2i, MapReduce, and full-text search) are useful but should be used judiciously. None of these scale nearly as well as key/value operations; queries that may work well on a few nodes in development may run more slowly on a busy development environment, particularly as the cluster grows in size.

Key/value operations seem primitive (and they are) but you'll find they are flexible, scalable, and very very fast (and predictably so).

Reads and writes in Riak should be as fast with ten billion values in storage as with ten thousand.

Design the main functionality of your application around the straight key/value operations that Riak provides and your software will continue to work at blazing speeds when you have petabytes of data stored across dozens of servers.

Normalization

Normalizating data is generally a useful approach in a relational database, but unlikely to lead to happy results with Riak.

Riak lacks foreign keys constraints and join operations, two vital parts of the normalization story, so reconstructing a single record from multiple objects would involve multiple read requests; certainly possible and fast enough on a small scale, but not ideal for larger requests.

Imagine the performance of your application if most of your requests involved a single read operation. Preparing and storing the answers to queries you're going to ask later is a best practice for Riak.

See [Denormalization] for more discussion.

Ducking conflict resolution

One of the first hurdles Basho faced when releasing Riak was educating developers on the complexities of eventual consistency and the need to intelligently resolve data conflicts.

Because Riak is optimized for high availability, *even when servers are offline or disconnected from the cluster due to network failures*, it is not uncommon for two servers to have different versions of a piece of data.

The simplest approach to coping with this is to allow Riak to choose a winner based on timestamps. It can do this more effectively if developers follow Basho's guidance on sending updates with *vector clock* metadata to help track causal history, but often concurrent updates cannot be automatically resolved via vector clocks, and trusting server clocks to determine which write was the last to arrive is a **terrible** conflict resolution method.

Even if your server clocks are magically always in sync, are your business needs well-served by blindly applying the most recent update? Some databases have no alternative but to handle it that way, but we think you deserve better.

Typed buckets in Riak 2.0 default to retaining conflicts and requiring the application to resolve them, but we're also providing replicated data types to automate conflict resolution on the servers.

If you want to minimize the need for conflict resolution, modeling with as much immutable data as possible is a big win.

[Conflict resolution] covers this in much more detail.

Mutability

For years, functional programmers have been singing the praises of immutable data, and it confers significant advantages when using a distributed data store like Riak.

Most obviously, conflict resolution is dramatically simplified when objects are never updated.

Even in the world of single-server database servers, updating records in place carries costs. Most databases lose all sense of history when data is updated, and it's entirely possible for two different clients to overwrite the same field in rapid succession leading to unexpected results.

Some data is always going to be mutable, but thinking about the alternative can lead to better design.

SELECT * FROM <table>

A perfectly natural response when first encountering a populated database is to see what's in it. In a relational database, you can easily retrieve a list of tables and start browsing their records.

As it turns out, this is a terrible idea in Riak.

Not only is Riak optimized for unstructured, opaque data, it is also not designed to allow for trivial retrieval of lists of buckets (very loosely analogous to tables) and keys.

Doing so can put a great deal of stress on a large cluster and can significantly impact performance.

It's a rather unusual idea for someone coming from a relational mindset, but being able to algorithmically determine the key that you need for the data you want to retrieve is a major part of the Riak application story.

Large objects

Because Riak sends multiple copies of your data around the network for every request, values that are too large can clog the pipes, so to speak, causing significant latency problems.

Basho generally recommends 1-4MB objects as a soft cap; larger sizes are possible with careful tuning, however.

We'll return to object size when discussing [Conflict resolution]; for the moment, suffice it to say that if you're planning on storing *mutable* objects in the upper ranges of our recommendations, you're particularly at risk of latency problems.

For significantly larger objects, Riak CS offers an Amazon S3-compatible (and also OpenStack Swift-compatible) key/value object store that uses Riak under the hood.

Running a single server

This is more of an operations anti-pattern, but it is a common misunderstanding of Riak's architecture.

It is quite common to install Riak in a development environment using its `devrel` build target, which creates 5 full Riak stacks (including Erlang virtual machines) to run on one server to simulate a cluster.

However, running Riak on a single server for benchmarking or production use is counterproductive, regardless of whether you have 1 stack or 5 on the box.

It is possible to argue that Riak is more of a database coordination platform than a database itself. It uses Bitcask or LevelDB to persist data to disk, but more importantly, it commonly uses *at least* 64 such embedded databases in a cluster.

Needless to say, if you run 64 databases simultaneously on a single filesystem you are risking significant I/O and CPU contention unless the environment is carefully tuned (and has some pretty fast disks).

Perhaps more importantly, Riak's core design goal, its raison d'être, is high availability via data redundancy and related mechanisms. Writing three copies of all your data to a single server is mostly pointless, both contributing to resource contention and throwing away Riak's ability to survive server failure.

Further reading

- Why Riak (docs.basho.com)
- Data Modeling (docs.basho.com)

- Clocks Are Bad, Or, Welcome to the Wonderful World of Distributed Systems (Basho blog)

Denormalization

Normal forms are the holy grail of schema design in the relational world. Duplication is misery, we learn. Disk space is constrained, so let foreign keys and join operations and views reassemble your data.

Conversely, when you step into a world *without* join operations, **stop normalizing**. In fact, go the other direction, and duplicate your data as much as you need to. Denormalize all the things!

I'm sure you immediately thought of a few objections to denormalization; I'll do what I can to dispel your fears. Read on, Macduff.

Disk space

Let me get the easy concern out of the way: don't worry about disk space. I'm not advocating complete disregard for it, but one of the joys of operating a Riak database is that adding more computing resources and disk space is not a complex, painful operation that risks downtime for your application or, worst of all, manual sharding of your data.

Need more disk space? Add another server. Install your OS, install Riak, tell the cluster you want to join it, and then pull the trigger. Doesn't get much easier than that.

Performance over time

If you've ever created a *really* large table in a relational database, you have probably discovered that your performance is abysmal. Yes, indexes help with searching large tables, but maintaining those indexes are **expensive** at large data sizes.

Riak includes a data organization structure vaguely similar to a table, called a *bucket*, but buckets don't carry the indexing overhead of a relational table. As you dump more and more data into a bucket, write (and read) performance is constant.

Performance per request

Yes, writing the same piece of data multiple times is slower than writing it once, by definition.

However, for many Riak use cases, writes can be asynchronous. No one is (or should be) sitting at a web browser waiting for a sequence of write requests to succeed.

What users care about is **read** performance. How quickly can you extract the data that you want?

Unless your application is receiving many hundreds or thousands of new pieces of data per second to be stored, you should have plenty of time to write those entries individually, even if you write them multiple times to different keys to make future queries faster. If you really *are* receiving so many objects for storage that you don't have time to write them individually, you can buffer and write blocks of them in chunks.

In fact, a common data pattern is to assemble multiple objects into larger collections for later retrieval, regardless of the ingest rate.

What about updates?

One key advantage to normalization is that you only have to update any given piece of data once.

However, many use cases that require large quantities of storage deal with mostly immutable data, such as log entries, sensor readings, and media storage. You don't change your sensor data after it arrives, so why do you care if each set of inputs appears in five different places in your database?

Any information which must be updated frequently should be confined to small objects that are limited in scope.

We'll talk much more about data modeling to account for mutable and immutable data.

Further reading

- NoSQL Data Modeling Techniques (Highly Scalable Blog)

Data modeling

It can be hard to think outside the table, but once you do, you may find interesting patterns that to use in any database, even relational.[1]

If you thoroughly absorbed the earlier content, some of this may feel redundant, but implications of the key/value model are not always obvious.

Rules to live by

As with most such lists, these are guidelines rather than hard rules, but take them seriously.

(1) Know your keys.

The cardinal rule of any key/value data store: the fastest way to get data is to know the key you want.

How do you pull that off? Well, that's the trick, isn't it?

The best way to always know the key you want is to be able to programmatically reproduce it based on information you already have. Need to know the sales data for one of your client's magazines in December 2013? Store it in a **sales** bucket and name the key after the client, magazine, and month/year combo.

Guess what? Retrieving it will be much faster than running a SQL `select` statement in a relational database.

And if it turns out that the magazine didn't exist yet, and there are no sales figures for that month? No problem. A negative response, especially for immutable data, is among the fastest operations Riak offers.

Because keys are only unique within a bucket, the same unique identifier can be used in different buckets to represent different information about the same entity (e.g., a customer address might be in an `address` bucket with the customer id as its key, whereas the customer id as a key in a `contacts` bucket would presumably contain contact information).

[1] Feel free to use a relational database when you're willing to sacrifice the scalability, performance, and availability of Riak… but why would you?

(2) Know your namespaces.

Riak has several levels of namespaces when storing data.

Historically, buckets have been what most thought of as Riak's virtual namespaces.

The newest level: **bucket types**, introduced in Riak 2.0, which group buckets for configuration and security purposes.

Less obviously, keys are their own namespaces. If you want a hierarchy for your keys that looks like `sales/customer/month`, you don't need nested buckets: you just need to name your keys appropriately, as discussed in (1). `sales` can be your bucket, while each key is prepended with customer name and month.

(3) Know your queries.

Writing data is cheap. Disk space is cheap.

Dynamic queries in Riak are very, very expensive.

As your data flows into the system, generate the views you're going to want later. That magazine sales example from (1)? The December sales numbers are almost certainly aggregates of smaller values, but if you know in advance that monthly sales numbers are going to be requested frequently, when the last data arrives for that month the application can assemble the full month's statistics for later retrieval.

Yes, getting accurate business requirements is non-trivial, but many Riak applications are version 2 or 3 of a system, written once the business discovered that the scalability of MySQL, Postgres, or MongoDB simply wasn't up to the job of handling their growth.

(4) Take small bites.

Remember your parents' advice over dinner? They were right.

When creating objects that will be updated, constrain their scope and keep the number of contained elements low to reduce the odds of multiple clients attempting to update the data concurrently.

(5) Create your own indexes.

Riak offers metadata-driven indexes (2i) and full-text indexes (Riak Search) for values, but these face scaling challenges: in order to identify all objects for a given index value, roughly a third of the cluster must be involved.

For many use cases, creating your own indexes is straightforward and much faster/more scalable, since you'll be managing and retrieving a single object.

See [Conflict resolution] for more discussion of this.

(6) Embrace immutability.

As we discussed in [Mutability], immutable data offers a way out of some of the challenges of running a high volume, high velocity data store.

If possible, segregate mutable from non-mutable data, ideally using different buckets for [request tuning][Request tuning].

Datomic is a unique data storage system that leverages immutability for all data, with Riak commonly used as a backend data store. It treats any data item in its system as a "fact," to be potentially superseded by later facts but never updated.

(7) Don't fear hybrid solutions.

As much we would all love to have a database that is an excellent solution for any problem space, we're a long way from that goal.

In the meantime, it's a perfectly reasonable (and very common) approach to mix and match databases for different needs. Riak is very fast and scalable for retrieving keys, but it's decidedly suboptimal at ad hoc queries. If you can't model your way out of that problem, don't be afraid to store keys alongside searchable metadata in a relational or other database that makes querying simpler, and once you have the keys you need, grab the values from Riak

Just make sure that you consider failure scenarios when doing so; it would be unfortunate to compromise Riak's availability by rendering it useless when your other database is offline.

Further reading

- Use Cases

Conflict resolution

Conflict resolution is an inherent part of nearly any Riak application, whether or not the developer knows it.

Conflict resolution strategies

There are basically 6 distinct approaches for dealing with conflicts in Riak, and well-written applications will typically use a combination of these strategies depending on the nature of the data.[2]

- Ignore the problem and let Riak pick a winner based on timestamp and context if concurrent writes are received (aka "last write wins").
- Immutability: never update values, and thus never risk conflicts.
- Instruct Riak to retain conflicting writes and resolve them with custom business logic in the application.
- Instruct Riak to retain conflicting writes and resolve them using client-side data types designed to resolve conflicts automatically.
- Instruct Riak to retain conflicting writes and resolve them using server-side data types designed to resolve conflicts automatically.

And, as of Riak 2.0, strong consistency can be used to avoid conflicts (but as we'll discuss below there are significant downsides to doing so).

[2] If each bucket has its own conflict resolution strategy, requests against that bucket can be tuned appropriately. For an example, see [Tuning for immutable data].

Last write wins

Prior to Riak 2.0, the default behavior was for Riak to resolve siblings by default (see [Tuning parameters] for the parameter `allow_mult`). With Riak 2.0, the default behavior changes to retaining siblings for the application to resolve, although this will not impact legacy Riak applications running on upgraded clusters.

For some use cases, letting Riak pick a winner is perfectly fine, but make sure you're monitoring your system clocks and are comfortable losing occasional (or not so occasional) updates.

Data types

It has always been possible to define data types on the client side to merge conflicts automatically.

With Riak 1.4, Basho started introducing distributed data types (formally known as **CRDTs**, or conflict-free replicated data types) to allow the cluster to resolve conflicting writes automatically. The first such type was a simple counter; Riak 2.0 adds sets and maps.

These types are still bound by the same basic constraints as the rest of Riak. For example, if the same set is updated on either side of a network split, requests for the set will respond differently until the split heals; also, these objects should not be allowed to grow to multiple megabytes in size.

Strong consistency

As of Riak 2.0, it is possible to indicate that values should be managed using a consensus protocol, so a quorum of the servers responsible for that data must agree to a change before it is committed.

This is a useful tool, but keep in mind the tradeoffs: writes will be slower due to the coordination overhead, and Riak's ability to continue to serve requests in the prence of network partitions and server failures will be compromised.

For example, if a majority of the primary servers for the data are unavailable, Riak will refuse to answer read requests if the surviving servers are not certain the data they contain is accurate.

Thus, use this only when necessary, such as when the consequences of conflicting writes are painful to cope with. An example of the need for this comes from Riak CS: because users are allowed to create new accounts, and because there's no convenient way to resolve username conflicts if two accounts are created at the same time with the same name, it is important to coordinate such requests.

Conflicting resolution

Resolving conflicts when data is being rapidly updated can feel Sysiphean.

It's always possible that two different clients will attempt to resolve the same conflict at the same time, or that another client will update a value between the time that one client retrieves siblings and it attempts to resolve them. In either case you may have new conflicts created by conducting conflict resolution.

Consider this yet another plug to consider immutability.

Further reading

- Clocks Are Bad, Or, Welcome to the Wonderful World of Distributed Systems (Basho blog)
- Index for Fun and for Profit (Basho blog)
- Readings in conflict-free replicated data types (Chris Meiklejohn's blog)

Request tuning

Riak is extensively (perhaps *too* extensively) configurable. Much of that flexibility involves platform tuning accessible only via the host operating system, but several core behavioral values can (and should) be managed by applications.

With the notable exceptions of `n_val` (commonly referred to as `N`) and `allow_mult`, the parameters described below can be overridden with each request. All of them can be configured per-bucket type (available with Riak 2.0) or per-bucket.

Key concepts

Any default value listed below as **quorum** is equivalent to `n_val/2+1`, or **2** whenever `n_val` has not been modified.

Primary servers are the cluster members that, in the absence of any network or server failure, are supposed to "own" any given key/value pair.

Riak's key/value engine does not itself write values to storage. That job is left to the **backends** that Riak supports: Bitcask, LevelDB, and Memory.

No matter what the parameters below are set to, requests will be sent to `n_val` servers on behalf of the client, **except** for strongly-consistent read requests with Riak 2.0, which can be safely retrieved from the current leader for that key/value pair.

Tuning parameters

Leave this alone

`n_val` The number of copies of data that are written. This is independent of the number of servers in the cluster. Default: **3**.

The `n_val` is vital to nearly everything that Riak does. The default value of 3 should never be lowered except in special circumstances, and changing it after a bucket has data can lead to unexpected behavior.

Configure at the bucket

`allow_mult` Specify whether this bucket retains conflicts for the application to resolve (`true`) or pick a winner using vector clocks and server timestamp even if the causality history does not indicate that it is safe to do so (`false`). See [Conflict resolution] for more. Default: `false` for untyped buckets (including all buckets prior to Riak 2.0), `true` otherwise

You **should** give this value careful thought. You **must** know what it will be in your environment to do proper key/value data modeling.

`last_write_wins` Setting this to `true` is a slightly stronger version of `allow_mult=false`: when possible, Riak will write new values to storage without bothering to compare against existing values. Default: `false`

Configure at the bucket or per-request

`r` The number of servers that must *successfully* respond to a read request before the client will be sent a response. Default: `quorum`

`w` The number of servers that must *successfully* respond to a write request before the client will be sent a response. Default: `quorum`

`pr` The number of *primary* servers that must successfully respond to a read request before the client will be sent a response. Default: **0**

`pw` The number of *primary* servers that must successfully respond to a write request before the client will be sent a response. Default: **0**

`dw` The number of servers that must respond indicating that the value has been successfully handed off to the *backend* for durable storage before the client will be sent a response. Default: **2** (effective minimum **1**)

`notfound_ok` Specifies whether the absence of a value on a server should be treated as a successful assertion that the value doesn't exist (`true`) or as an error that should not count toward the `r` or `pr` counts (`false`). Default: `true`

Impact

Generally speaking, the higher the integer values listed above, the more latency will be involved, as the server that received the request will wait for more servers to respond before replying to the client.

Higher values can also increase the odds of a timeout failure or, in the case of the primary requests, the odds that insufficient primary servers will be available to respond.

Write failures

Please read this. Very important. Really.

The semantics for write failure are *very different* under eventually consistent Riak than they are with the optional strongly consistent writes available in Riak 2.0, so I'll tackle each separately.

Eventual consistency

In most cases when the client receives an error message on a write request, *the write was not a complete failure.* Riak is designed to preserve your writes whenever possible, even if the parameters for a request are not met. **Riak will not roll back writes.**

Even if you attempt to read the value you just tried to write and don't find it, that is **not** definitive proof that the write was a complete failure. (Sorry.)

If the write is present on at least one server, *and* that server doesn't crash and burn, *and* future updates don't supersede it, the key and value written should make their way to all servers responsible for them.

Retrying any updates that resulted in an error, with the appropriate vector clock to help Riak intelligently resolve conflicts, won't cause problems.

Strong consistency

Strong consistency is the polar opposite from the default Riak behaviors. If a client receives an error when attempting to write a value, it is a safe bet that the value is not stashed somewhere in the cluster waiting to be propagated, **unless** the error is a timeout, the least useful of all possible responses.

No matter what response you receive, if you read the key and get the new value back[3], you can be confident that all future successful reads (until the next write) will return that same value.

Tuning for immutable data

If you constrain a bucket to contain nothing but immutable data, you can tune for very fast responses to read requests by setting `r=1` and `notfound_ok=false`.

This means that read requests will (as always) be sent to all `n_val` servers, but the first server that responds with **a value other than** `notfound` will be considered "good enough" for a response to the client.

Ordinarily with `r=1` and the default value `notfound_ok=true` if the first server that responds doesn't have a copy of your data you'll get a `not found` response; if a failover server happens to be actively serving requests, there's a very good chance it'll be the first to respond since it won't yet have a copy of that key.

Further reading

- Buckets (docs.basho.com)
- Eventual Consistency (docs.basho.com)
- Replication (docs.basho.com)
- Understanding Riak's Configurable Behaviors (Basho blog series)

[3] To be *absolutely certain* your value is in Riak after a write error and a successful read, you can issue a new read request not tied to any existing object; your client library could be caching the value you just wrote.

Chapter 6

Notes

A Short Note on RiakCS

Riak CS is Basho's open source extension to Riak to allow your cluster to act as a remote storage mechanism, comparable to (and compatible with) Amazon's S3. There are several reasons you may wish to host your own cloud storage mechanism (security, legal reasons, you already own lots of hardware, cheaper at scale). This is not covered in this short book, though I may certainly be bribed to write one.

A Short Note on MDC

MDC, or Multi Data Center, is a commercial extension to Riak provided by Basho. While the documentation is freely available, the source code is not. If you reach a scale where keeping multiple Riak clusters in sync on a local or global scale is necessary, I would recommend considering this option.

Locks, a cautionary tale

While assembling the *Writing Applications* guide, I tried to develop a data model that would allow for reliable locking without strong consistency. That attempt failed, but rather than throw the idea away entirely, I decided to include it here to illustrate the complexities of coping with eventual consistency.

Basic premise: multiple workers may be assigned data sets to process, but each data set should be assigned to no more than one worker.

In the absence of strong consistency, the best an application can do is use the `pr` and `pw` parameters (primary read, primary write) with a value of `quorum` or `n_val`.

So, common features to all of these models:

- Lock for any given data set is a known key
- Value is a unique identifier for the worker

Lock, a first draft

Sequence

Bucket: `allow_mult=false`

1. Worker reads with `pr=quorum` to determine whether a lock exists
2. If it does, move on to another data set
3. If it doesn't, create a lock with `pw=quorum`
4. Process data set to completion
5. Remove the lock

Failure scenario

1. Worker #1 reads the non-existent lock
2. Worker #2 reads the non-existent lock
3. Worker #2 writes its ID to the lock
4. Worker #2 starts processing the data set
5. Worker #1 writes its ID to the lock
6. Worker #1 starts processing the data set

Lock, a second draft

Bucket: `allow_mult=false`

Sequence

1. Worker reads with `pr=quorum` to determine whether a lock exists
2. If it does, move on to another data set
3. If it doesn't, create a lock with `pw=quorum`
4. Read lock again with `pr=quorum`
5. If the lock exists with another worker's ID, move on to another data set
6. Process data set to completion
7. Remove the lock

Failure scenario

1. Worker #1 reads the non-existent lock
2. Worker #2 reads the non-existent lock
3. Worker #2 writes its ID to the lock
4. Worker #2 reads the lock and sees its ID
5. Worker #1 writes its ID to the lock
6. Worker #1 reads the lock and sees its ID
7. Both workers process the data set

If you've done any programming with threads before, you'll recognize this as a common problem with non-atomic lock operations.

Lock, a third draft

Bucket: `allow_mult=true`

Sequence

1. Worker reads with `pr=quorum` to determine whether a lock exists
2. If it does, move on to another data set
3. If it doesn't, create a lock with `pw=quorum`
4. Read lock again with `pr=quorum`
5. If the lock exists with another worker's ID **or** the lock contains siblings, move on to another data set
6. Process data set to completion
7. Remove the lock

Failure scenario

1. Worker #1 reads the non-existent lock
2. Worker #2 reads the non-existent lock
3. Worker #2 writes its ID to the lock
4. Worker #1 writes its ID to the lock
5. Worker #1 reads the lock and sees a conflict
6. Worker #2 reads the lock and sees a conflict
7. Both workers move on to another data set

Lock, a fourth draft

Bucket: `allow_mult=true`

Sequence

1. Worker reads with `pr=quorum` to determine whether a lock exists
2. If it does, move on to another data set
3. If it doesn't, create a lock with `pw=quorum` and a timestamp
4. Read lock again with `pr=quorum`
5. If the lock exists with another worker's ID **or** the lock contains siblings **and** its timestamp is not the earliest, move on to another data set
6. Process data set to completion
7. Remove the lock

Failure scenario

1. Worker #1 reads the non-existent lock
2. Worker #2 reads the non-existent lock
3. Worker #2 writes its ID and timestamp to the lock
4. Worker #2 reads the lock and sees its ID
5. Worker #2 starts processing the data set
6. Worker #1 writes its ID and timestamp to the lock
7. Worker #1 reads the lock and sees its ID with the lowest timestamp
8. Worker #1 starts processing the data set

At this point I may hear you grumbling: clearly worker #2 would have the lower timestamp because it attempted its write first, and thus #1 would skip the data set and try another one.

Even *if* both workers are running on the same server (and thus *probably* have timestamps that can be compared)[1], perhaps worker #1 started its write earlier but contacted an overloaded cluster member that took longer to process the request.

The same failure could occur if, instead of using timestamps for comparison purposes, the ID of each worker was used for comparison purposes. All it takes is for one worker to read its own write before another worker's write request arrives.

[1] An important part of any distributed systems discussion is the fact that clocks are inherently untrustworthy, and thus calling any event the "last" to occur is an exercise in faith: faith that a system clock wasn't set backwards in time by `ntpd` or an impatient system administrator, faith that all clocks involved are in perfect sync.

And, to be clear, perfect synchronization of clocks across multiple systems is unattainable. Google is attempting to solve this by purchasing lots of atomic and GPS clocks at great expense, and even that only narrows the margin of error.

Conclusion

We can certainly come up with algorithms that limit the number of times that multiple workers tackle the same job, but I have yet to find one that guarantees exclusion.

What I found surprising about this exercise is that none of the failure scenarios required some of the odder edge conditions that can cause unexpected outcomes. For example, `pw=quorum` writes will return an error to the client if 2 of the primary servers are not available, but the value will *still* be written to the 3rd server and 2 fallback servers. Predicting what will happens the next time someone tries to read the key is challenging.

None of these algorithms required deletion of a value, but that is particularly fraught with peril. It's not difficult to construct scenarios where deleted values reappear if servers are temporarily unavailable during the deletion request.