Why Riak

Contents

1. What is Riak?
2. How Does a Riak Cluster Work?
3. When Things Go Wrong

What is Riak?

Riak is a distributed database designed to deliver maximum data availability by distributing data across multiple servers. As long as your client can reach *one* Riak server, it should be able to write data. In most failure scenarios, the data you want to read should be available, although it may not be the most up-to-date version of that data.

This fundamental tradeoff—high availability in exchange for possibly outdated information—informs the key architectural decisions behind Riak. This idea of "eventual consistency" is a common one in distributed systems, with DNS and web caches as two notable examples.

Basho's goals for Riak

Goal	Description
Availability	Riak writes to and reads from multiple servers to offer

	data availability even when hardware or the network itself are experiencing failure conditions
Operational simplicity	Easily add new machines to your Riak cluster without incurring a larger operational burden
Scalability	Riak automatically distributes data around the cluster and yields a near-linear performance increase as you add capacity
Masterless	Your requests are not held hostage to a specific server in the cluster that may or may not be available

When Riak makes sense

If your data does not fit on a single server and demands a distributed database architecture, then you should absolutely take a close look at Riak as a potential solution to your data availability issues. Getting distributed databases right is **very** difficult, and Riak was built to address data availability issues with as few trade-offs and downsides as possible.

In essence, Riak's focus on availability makes it a good fit whenever downtime is unacceptable. No one can promise 100% uptime, but Riak is designed to survive network partitions and hardware failures that would significantly disrupt most databases.

A less-heralded feature of Riak is its predictable latency. Because its fundamental operations—read, write, and delete—do not involve complex data joins or locks, it services those requests promptly. Thanks to this capability Riak is often selected as a data storage backend for other data management software from a variety of paradigms.

When Riak is Less of a Good Fit

Basho recommends that you run no fewer than 5 data servers in a cluster. This means that Riak can be overkill for small databases. If you're not already sure that you will need a distributed database, there's a good chance that you won't need Riak.

Nonetheless, if explosive growth is a possibility, you are always highly advised to prepare for that in advance. Scaling at Internet speeds is sometimes compared to overhauling an airplane mid-flight. If you feel that such a transition might be necessary in the future, then you might want to consider Riak.

Riak's simple data model, consisting of keys and values as its atomic elements, means that your data must be denormalized if your system is to be reasonably performant. For most applications this is not a serious hurdle. But if your data simply cannot be effectively managed as keys and values Riak will most likely not be the best fit for you.

On a related note: while Riak offers ways to find values that match certain criteria, if your application demands a high query load by any means other than the keys—e.g. SQL-style `SELECT * FROM table` operations—Riak will not be as efficient as other databases. If you wish to compare Riak with other data technologies, Basho offers a tool called **basho_bench** to help measure its performance so that you can decide whether the availability and operational benefits of Riak outweigh its disadvantages.

How Does a Riak Cluster Work?

What is a Riak Node?

A Riak node is not quite the same as a server, but in a production environment the two should be equivalent. A developer may run multiple nodes on a single laptop, but this would never be advisable in a real production cluster.

Each node in a Riak cluster is equivalent, containing a complete, independent copy of the whole Riak package. There is no “master.” No node has more responsibilities than others, and no node has special tasks not performed by other nodes. This uniformity provides the basis for Riak's fault tolerance and scalability.

Each node is responsible for multiple data partitions as discussed below.

Riak Automatically Re-Distributes Data When Capacity is Added

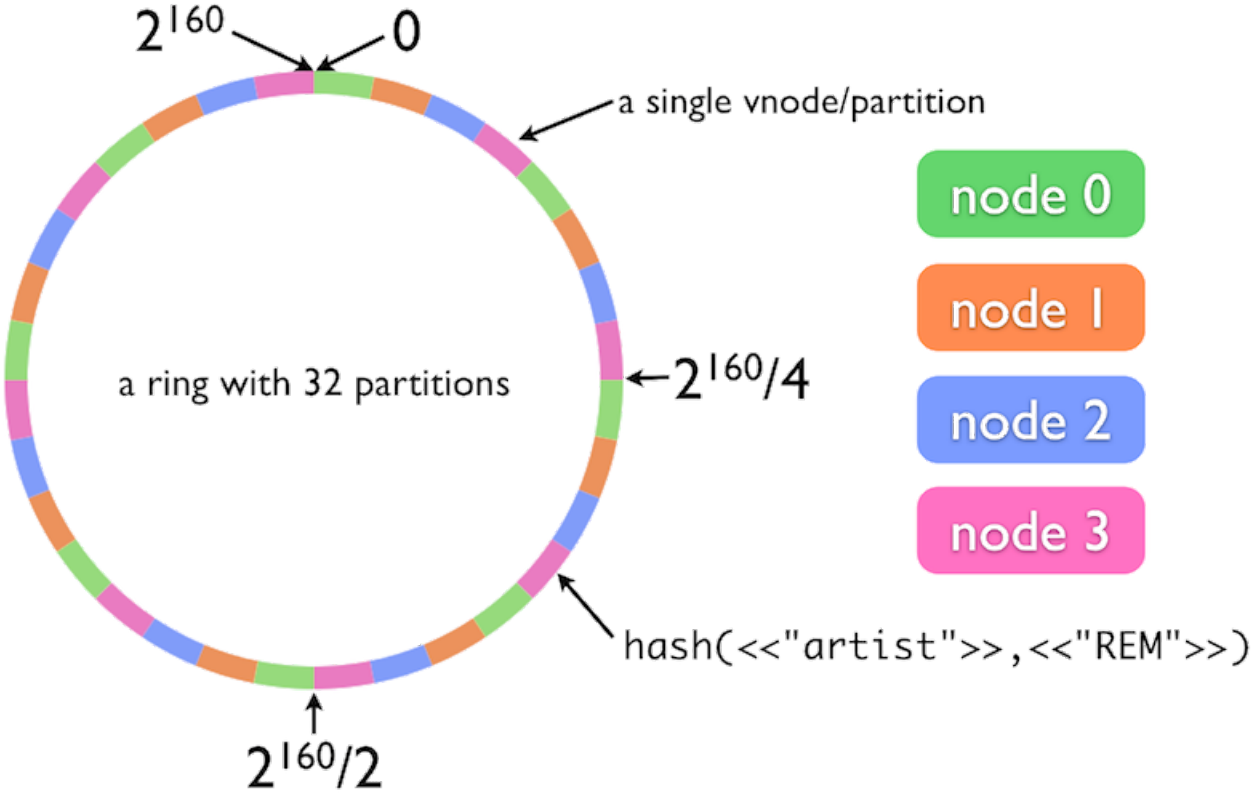
When you add (or remove) machines, data is rebalanced automatically with no downtime. New machines claim data until ownership is equally spread around the cluster, with the resulting cluster status updates shared to every node via a gossip protocol and used to route requests. This is what makes it possible for any node in the cluster to receive requests. The end result is that developers don't need to deal with the underlying complexity of where data lives.

Consistent Hashing

Data is distributed across nodes using consistent hashing. Consistent hashing ensures that data is evenly distributed around the cluster and makes possible the automatic redistribution of data as the cluster scales.

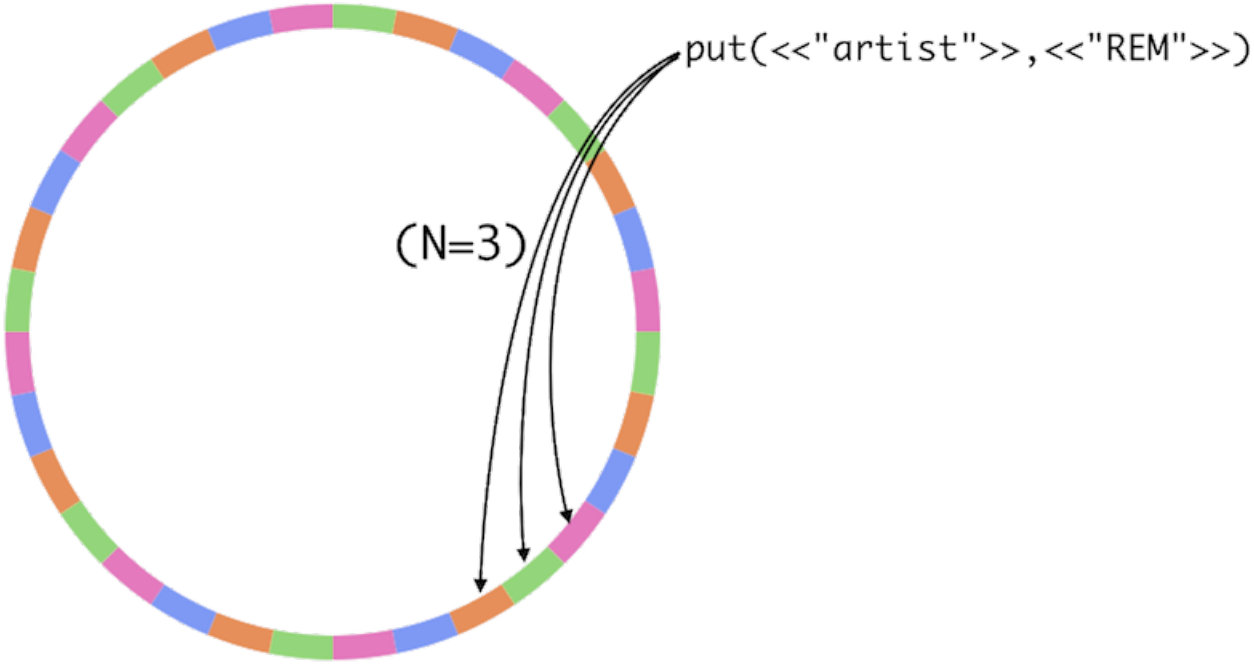
How does consistent hashing work? Riak stores data using a simple key/value scheme. These keys are associated with a namespace called a **bucket**. When you perform key/value operations in Riak, the bucket and key combination is hashed. The resulting hash maps onto a 160-bit integer space. You can think of this integer space as a ring used to determine what data to put on which physical machines.

How is this determination made? Riak divides the integer space into equally-sized partitions. Each partition owns a range of values on the ring, and is responsible for all buckets and keys that, when hashed, fall into that range. Each partition is managed by a process called a virtual node (or **vnode***). Physical machines evenly divide responsibility for vnodes. Let's say that you have a 4-node cluster with 32 partitions managed by 32 vnode processes. Each of the four physical machines claim eight vnodes (as illustrated below). Each physical machine thus becomes responsible for all keys represented by its eight vnodes.



Intelligent Replication

Riak's replication scheme ensures that you can still read, write, and update data if nodes go down. Riak allows you to set a replication variable, *n*. An *n* value of 3 (the default) means that each object is replicated 3 times. When an object's key is mapped onto a given partition, Riak won't stop there: it will automatically replicate the data onto the next two partitions as well.



When Things Go Wrong

Riak retains fault tolerance, data integrity, and availability even in failure conditions such as hardware failure and network partitions. Riak has a number of means of addressing these scenarios and other bumps in the road, like version conflicts in data.

Hinted Handoff

Hinted handoff enables Riak to handle node failure. If a node goes down, a neighboring node will take over its storage operations. When the failed node returns, the updates received by the neighboring node are handed back to it. This ensures availability for writes and updates and happens automatically, minimizing the operational burden of failure conditions.

Version Conflicts

In any system that replicates data, conflicts can arise, for example when two clients update the same object at the exact same time or when not all updates have yet

reached hardware that is experiencing lag. Furthermore, in Riak, replicas are "eventually consistent," meaning that while data is always available, not all replicas may have the most recent update at the exact same time, causing brief periods—generally on the order of milliseconds—of inconsistency while all state changes are synchronized.

How is this divergence addressed? When you make a read request, Riak looks up all replicas for that object. By default, Riak will return the most recently updated version, determined by looking at the object's vector clock. Vector clocks are metadata attached to each replica when it is created. They are extended each time a replica is updated to keep track of versions. You can also allow clients to resolve conflicts themselves if that is a better fit for your use case.

Read Repair

When an outdated replica is returned as part of a read request, Riak will automatically update the out-of-sync replica to make it consistent. Read repair, a self-healing property of the database, will even update a replica that returns a `not_found` in the event that a node loses the data due to physical failure.

Reading and Writing Data in Failure Conditions

In Riak, you can set an *r* value for reads and a *w* value for writes. These values give you control over how many replicas must respond to a request for it to succeed. Let's say that you have an *n* value of 3, but one of the physical nodes responsible for a replica is down. With an r=2 setting, only 2 replicas must return results for read to be deemed successful. This allows Riak to provide read availability even when nodes are down or laggy. The same applies for the *w* in writes. If this value is not specified, Riak defaults to **quorum**, according to which the majority of nodes must respond. There is more on Replication Properties elsewhere in the documentation.