basho

www.basho.com

From Relational to Riak

Table of Contents

Table of Contents 1
Introduction 1
Why Migrate to Riak? 1
The Requirement of High Availability 1
Minimizing the Cost of Scale 2
Simple Data Models 4
Tradeoff Decisions 5
Eventual Consistency 5
Data Modeling 5
Operational & Development Considerations 6
Data Migration 6
The Basics of Modeling Data In Riak 8
Resolving Data Conflicts 8
Multi-Datacenter Operations 9
Conclusion 10

Introduction

This technical brief is designed to provide a background and detailed level of understanding for those analyzing a move from a relational database to a NoSQL model (specifically emphasizing the Riak key/value store offered by Basho).

The brief begins by examining commonly cited reasons for choosing Riak instead of a relational database, specifically focused on availability versus consistency tradeoffs, scalability, and the key/value data model. Then it analyzes the decision points that should be considered when choosing a non-relational solution and what such offerings cannot provide related to querying, data modeling, and consistency guarantees. Finally, it provides simple patterns for building common applications in Riak using its key/value design; dealing with data conflicts that emerge in an eventually consistent system; and how replicating data to multiple sites is possible with Riak.

Why Migrate to Riak?

This section analyzes the most common reasons to move from a relational database to Riak.

The Requirement of High Availability

Relational Databases tend to favor consistency over availability, making them ill suited for applications that require high availability

The CAP theorem, fathered by Dr. Eric Brewer, states that in the event of a network partition, a distributed system can either provide availability or consistency. Consistent operations provide applications with guarantees that read operations reflect the last successful update to the database, and are an important facet of enabling operations, like transactions, that are essential for some types of applications. Billing and financial systems are, typically, the canonical example used when referring to strongly consistent operations. However, in a piece entitled *CAP Twelve Years Later: How the "Rules" Have Changed*, Dr. Eric Brewer explains that banking systems depend "not on consistency for correctness, but rather on auditing and compensation."

Consistency is relatively straightforward when systems are constrained to a single server, which is a common configuration for traditional relational database management systems (RDBMS). If the dataset grows beyond the capacity of a single machine it becomes necessary to scale the database and operate in a distributed environment. Relational databases typically address the challenge of scale with a master/slave architecture, wherein the topology of a cluster is comprised of a single master node and multiple slaves. Under this configuration, the master node is responsible for accepting all write operations and coordinating with slave nodes to apply the updates in a consistent manner. Read requests can either be proxied through the master or sent directly to a slave.

However, in the event that a master node fails, the database will favor consistency and reject write operations until the failure is resolved. This can lead to a window of write unavailability, which is unacceptable in some application designs. Most master/slave architectures recognize that a master node is a single point of failure and

will perform automatic master failover, wherein a slave will be elected as a new master when failure of the master node is detected.

In contrast, Riak is a masterless system designed to favor availability, even in the event of node failures and/or network partitions. Any server ("node" in Riak parlance) can serve any incoming request, regardless of data locality, and all data is replicated across multiple nodes. If a node experiences an outage, other nodes will continue to service write and read requests. Further, if a node becomes unavailable to the rest of the cluster, a neighboring node will take over write and update responsibilities for the missing node. The neighboring node will pass new or updated data (termed "objects") back to the original node once it rejoins the cluster through a process called "hinted handoff."

Riak's masterless design ensures read and write availability is maintained even if many nodes become unavailable due to network partition or hardware failure. However, a lack of master/slave configuration to ensure consistency means that data in Riak is eventually consistent – all updates will eventually propagate to all nodes.

For many of today's application and platforms, high availability is more important than strict consistency. In some use cases, data unavailability can have a direct impact on revenue – cloud services, online retail, shopping carts, checkout process, advertising are just a few examples. Further, lack of availability can damage user trust and result in a poor user experience for many websites, social, and mobile applications; and for critical data, like user data or serving content, availability can be more important than strict consistency.

Minimizing the Cost of Scale

Scaling a relational database to handle more data and usage can be prohibitively expensive for operators.

Distributing data across several database servers to achieve scale often utilizes a technique called "sharding." A common example of this would be putting user data for differing geographical regions (e.g., US and EU) on different machines, or using an alphabetical or numerical order to split data.

Using business rules to shard data can be problematic for several reasons. First, writing and maintaining custom sharding logic increases the overhead of operating and developing an application on the database. Significant growth of data or traffic typically means significant, often manual, resharding projects. Determining how to intelligently split the dataset without negatively impacting performance, operations, and development presents a substantial challenge - especially when dealing with "big data," rapid scale, or peak loads. Further, rapidly growing applications frequently outpace an existing sharding scheme. When the data in a shard grows too large, the shard must again be split. While several "auto"-sharding technologies have emerged in recent years, these methods are often imprecise and manual intervention is standard practice. Finally, sharding can often lead to "hot spots" in the database – physical machines responsible for storing and serving a disproportionately high amount of both data and requests – which can lead to unpredictable latency and degraded performance.

In Riak, data is automatically distributed evenly across nodes using consistent hashing. Consistent hashing ensures data is evenly distributed around the cluster and new nodes can be added with automatic, minimal

reshuffling of data. This significantly decreases risky "hot spots" in the database and lowers the operational burden of scaling.

Riak stores data using a simple key/value model. Data entries in Riak are referred to as "objects." Key/value pairs are logically grouped together in a namespace called a bucket. As you write new keys to Riak, the object's bucket/key pair is hashed. The resulting value maps onto a 160-bit integer space. This integer space can be conceptualized as a ring that is used to determine where data is placed on physical machines.

Riak tokenizes the total key space into a fixed number of equally sized partitions (default is 64). Each partition owns the given range of values on the ring and is responsible for all buckets and keys that, when hashed, fall into that range. A virtual node (a "vnode") is the process that manages each of these partitions. Physical machines evenly divide responsibility for vnodes.

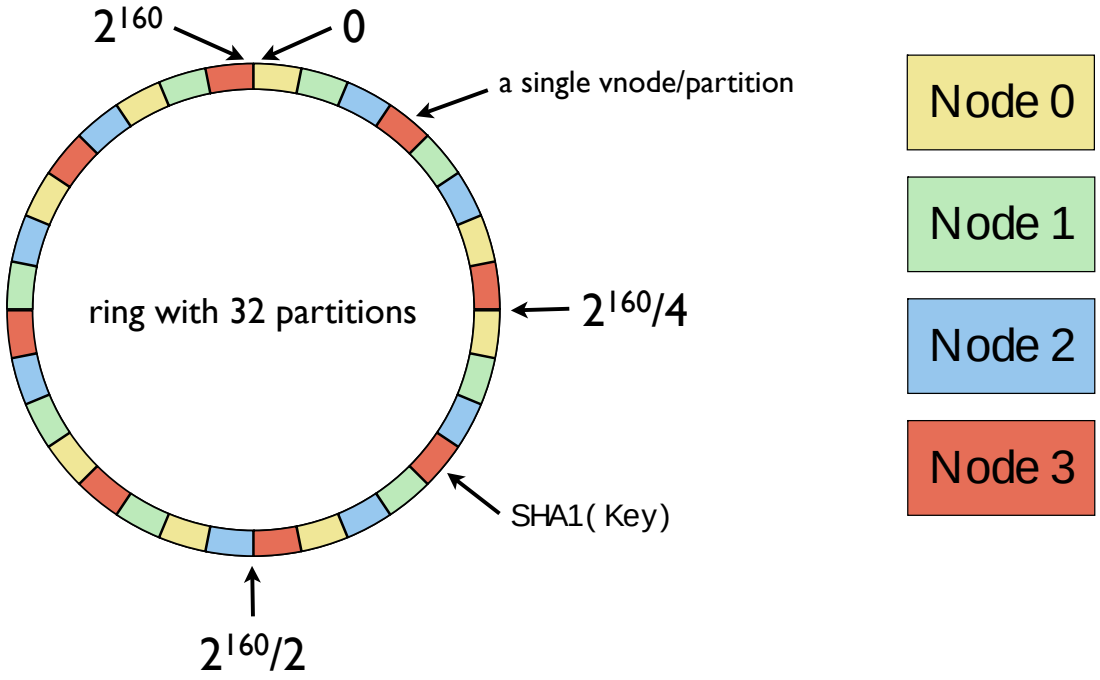

Figure 1: The Riak "Ring"

Hashing and shared responsibility for keys across nodes ensures that data in Riak is evenly distributed. When machines are added, data is rebalanced automatically with no downtime. New machines take responsibility for their share of data by assuming ownership of some of the partitions; existing cluster members hand off the relevant partitions and the associated data. The new node continues claiming partitions until data ownership is equal. This current cluster state is shared to every node using a gossip protocol and serves as a guide for routing requests. This process is what ensures that any node in the cluster is able to receive requests, taking routing concerns out of developers' hands.

For Riak users, consistent hashing, and the consequential flexible capacity handling, provides a much simpler operational scenario than manually sharding data.

Simple Data Models

The relational data model can be needlessly complex and inflexible for certain types of applications.

The data models of traditional relational databases offer many features that developers find important. However, with the emergence of trends like big data, "agile" development, and new types of applications (social and mobile), there has also been an increasing desire to store unstructured data, data that does not require the rigid data model of relational systems.

Many applications can effectively utilize a simple key/value model for storing and retrieving data. In Riak, objects are comprised of key/value pairs, which are stored in flat namespaces called "buckets." Riak does not dictate what types of data are persisted – all objects are stored on disk as binaries. As a result, developing code that interacts with this simple, straightforward design can be accomplished more efficiently. Additionally, adding new features to the application will not require updating a scheme or changing the data model, ideal for applications where rapid iterations are required and changes in the underlying data model are undesirable. A key/value design is not appropriate for all applications and use cases, but for many this model provides more flexibility and simplicity, thereby contributing to developer productivity. Later in this technical brief, some common use cases for Riak, and approaches to modeling data for those use cases, are discussed.

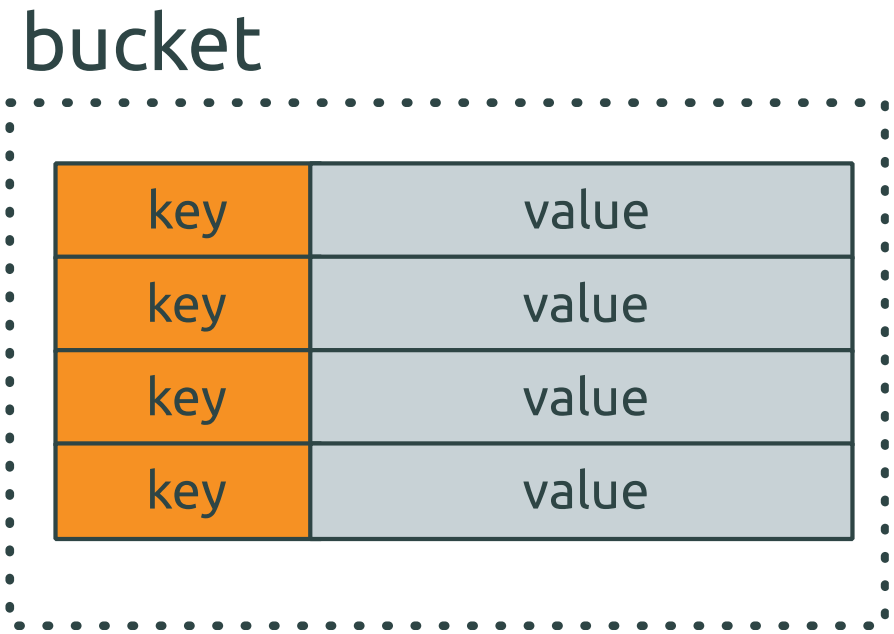

Figure 2: Key/Value Pairs Stored in a Bucket

Tradeoff Decisions

Eventual Consistency

Riak does not support strictly consistent operations

In a distributed and fault-tolerant environment like Riak, the unexpected is expected. That means that nodes may leave and join the cluster at any time, be it by accident (node failure, network partitions, etc) or administratively for cluster maintenance. Even with one or more nodes unreachable, the cluster is still expected to accept new writes and serve reads. Furthermore, the system is expected to return the same data from all nodes *eventually,* even from the failed nodes after the rejoin the cluster. This is made possible by mechanisms such as hinted handoff and read-repair.

Eventual consistency in Riak may come as bit of a surprise at first. Riak has no locks, allows simultaneous writes, and provides no guarantees over the ordering of concurrent operations. For many use cases, the ability for multiple writers to update the same value does not pose a threat, for instance if your application is storing profile data, which should only be updated by a single user. In other cases, simultaneous updates to the same object can cause conflicts, which must be resolved. While there are mechanisms such as Vector Clocks to help deal with these issues, if your application requires the kind of strong consistency found in ACID systems, Riak may not be a good fit.

Resiliency parameters can be tuned on a per-bucket or per-request basis. By using quorums, users can set an *r* and *w* value that adjusts the number of replicas that must respond to a read or write request for it to succeed. For example, if Riak has been configured to store three replicas of each object, a request with an *r* value of 2 will only require two out of the three replicas to respond before it is considered successful. In this scenario, Riak will be able to withstand a single node failure and still operate as expected. However, should another node fail simultaneously, this same request would fail to satisfy the quorum and not return an object.

To learn more, review our documentation and read our blog series entitled *Understanding Riak's Configurable Behaviors*.

Data Modeling

Riak's design does not support the richer data types and model of traditional relational systems

While many users find that Riak's key/value model is more flexible, faster to develop against, and well suited to their applications, there are tradeoffs regarding query options and data types available. Riak does not expose sets, counters, or transactions; it does not support join operations as there is no concept of columns and rows. Riak is queried via HTTP requests, via the protocol buffers API, or through various client libraries; there is no SQL or SQL-like language. Riak's simpler data model results in fewer and leaner query ability options.

Riak does, however, offer additional functionality on top of the fundamental key/value model:

- **Riak Search:** Riak Search is a distributed, full-text search engine. It provides support for various MIME types & analyzers, and robust querying including exact matches, wildcards, range queries, proximity searches, and more.
- **Secondary Indexing:** Secondary Indexing (2i) in Riak gives developers the ability to tag an object stored in Riak with one or more queryable values. Indexes can be either integers or strings, and can be queried by either exact matches or ranges of an index.
- **MapReduce:** Developers can leverage Riak MapReduce for tasks like filtering documents by tag, counting words in documents, and extracting links to related data. It offers support for JavaScript and Erlang.

For more information, check out the Riak documentation on Querying Data. Additionally, work is being done to expose additional data types that are tolerant of Riak's eventually consistent nature, specifically counters and sets, which will be publicly available soon.

Operational & Development Considerations

Data Migration

How should data be migrated to Riak? The topic of data migration could fill an entire book, so these are just a few points of starting advice. Should you desire in-depth help or consultation, the Professional Services team at Basho is always available for assistance.

The recommended method of migration is to adopt a staged approach, migrating areas of the application to Riak while running it alongside the previous data storage mechanism. For each stage:

1. Pick a standalone logical unit of data (a group of related tables, or a document in the current storage system).
2. Convert it to a storage format appropriate to Riak (the columns of a relational table map easily into JSON or XML fields) and consider how the data will be accessed (plain key/value reads and writes, or more complex queries).
3. Create migration scripts.

Most applications will have standalone high-traffic areas that are perfect to model as key/value storage operations, such as sessions, user preferences, advertisements, small binary or XML/JSON documents. A sure sign of these areas is that the applications gets the keys "for free," without having to perform any queries to discover them. For example, a user logs in and is assigned a session cookie – the application now has two easy keys in memory (a user id and session id) with which to load data. Or, a web request comes in and returns a content or ad id as part of the URL.

If a relational database is already in use, these will be the objects that are loaded from a single table or from several tightly related ones. They will, likely, have already been optimized for high read traffic, possibly de-

normalized from several related tables, for ease and speed of access. If possible, postpone any area that requires atomic multi-step transactions as the first point of migration. The topic of consistency and collision handling in highly parallel distributed systems is an advance topic that requires a deep understanding of the workings of Riak and the capabilities it provides.

Once a slice of the data model has been isolated for migration, consider what key and object format will be used. In most cases, the keys will be dictated by the existing application data (the format of the session id or user id will be already be defined), and these objects can be reused as Riak object keys. The format of the object payload requires additional consideration. If small binaries are being stored (PDF documents, small images, or custom binary data objects), these can be stored directly as binary blobs. If structured data is being migrated (for example, relational database tables), consider using structured text documents such as JSON or XML. If metadata is required alongside the object (timestamps, owner ids, tags of any kind), consider whether to store it as custom Riak object headers or as additional fields in the JSON/XML object payload.

Having made these design choices, the migration becomes fairly straightforward. Code must be written in a preferred programming language to retrieve the data from an existing system, compose it into appropriate Riak objects with the keys and values defined during analysis, and upload the results to the Riak cluster. There is no automated or "backend" way to migrate the data into Riak; the only way to ensure data is properly stored, and the appropriate indexes are created (in the case of Search or Secondary Index functionality), is to use the Riak client API to perform the writes.

After migration of the obvious areas perfect for simple key/value reads and writes are complete, it is likely desirable to migrate the data that requires more advanced querying capability or complex one-to-many and many-to-many relationships. Basho documentation and tutorials are valuable resources for data modeling and use cases, Key Filtering, Secondary Indexes, Search, and Map/Reduce queries. While there is an initial learning curve when transitioning from a traditional relational model to a distributed key/value system, and there is no exact equivalent to the familiar table join, rest assured that Riak does offer powerful and flexible data modeling and querying capabilities that have been field-tested on a massive scale. Basho, and the broader Riak community, provide helpful information via the Riak-users mailing list, or contact Basho directly for advice, code and architecture review, and consultation.

The Basics of Modeling Data In Riak

The chart below illustrates key/value mappings for common application types. Remember that values in Riak are opaque and stored on disk as binaries – JSON or XML documents, images, text, etc. The way data is organized in Riak should take into account the unique needs of the application, including access patterns such as read/write distribution, latency differences between various operations, use of Riak features (including MapReduce, Search, Secondary Indexes), and more.

Application Type	Key	Value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content
Logs	Date	Log File
Sensor	Date, Date/Time	Sensor Updates
User Data	Login, eMail, UUID	User Attributes
Content	Title, Integer	Text, JSON/XML/HTML Document, Images, etc.

For additional information, and more complex considerations such as modeling relationship and advanced social applications, see the Riak documentation on use cases and data modeling.

Resolving Data Conflicts

In any system that replicates data, conflicts can arise – e.g., if two clients update the same object at the exact same time or if not all updates have yet reached hardware that is experiencing lag. As discussed earlier, Riak is "eventually consistent" – while data is always available, not all replicas may have the most recent update at the same time, causing brief periods (generally on the order of milliseconds) of inconsistency while all state changes are synchronized.

However, Riak does provide features to detect and help resolve the statistically small number of incidents when data conflicts occur. When a read request is performed, Riak looks up all replicas for that object. By default, Riak will return the most 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. Clients can also be allowed to resolve conflicts themselves.

Further, when an outdated object is discovered 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 it due to physical failure.

Riak also features "Active Anti-Entropy," which is an automatic self-healing property that runs in the background. Rather than waiting for a read request to trigger a replica repair (as with Read Repair), Active Anti-

Entropy constantly uses a hash tree exchange to compare replicas of objects and automatically repairs or updates any that are divergent, missing, or corrupt. This can be beneficial for large clusters storing "stale" data.

More information on vector clocks and conflict resolution can be found in the online documentation.

Multi-Datacenter Operations

Multi-site replication is quickly becoming critical for many of today's platforms and applications. Not only does replication across multiple clusters provide geographic data locality – the ability to serve global traffic at low-latencies - it can also be an integral part of a disaster recovery or backup strategy. Other teams may use multi-site replication to maintain secondary data stores, both for failover as well as for performing intensive computation without disrupting production load. Multi-site replication is included in Basho's commercial extension to Riak, Riak Enterprise, which also includes 24/7 support.

Multi-site replication in Riak works differently than the typical approach seen in the relational world, multi-master replication. In Riak's multi-datacenter replication, one cluster acts as a "primary cluster." The primary cluster handles replication request from one or more "secondary clusters" (generally located in datacenters in other regions or countries). If the datacenter with the primary cluster goes down, a secondary cluster can take over as the primary cluster. In this sense, Riak's multi-datacenter capabilities are "masterless."

In multi-datacenter replication, there are two primary modes of operation: full sync and real-time. In full sync mode, a complete synchronization occurs between primary and secondary cluster(s). In real-time mode, continual, incremental synchronization occurs – replication is triggered by new updates. Full sync is performed upon initial connection of a secondary cluster, and then periodically (by default, every 6 hours). Full sync is also triggered if the TCP connection between primary and secondary clusters is severed and then recovered.

Data transfer is unidirectional (primary->secondary). However, bidirectional synchronization can be achieved by configuring a pair of connections between clusters.

Full documentation for multi-datacenter replication in Riak Enterprise is available in the online documentation.

Conclusion

Picking the right database for your team means a careful understanding of the requirements of your application or platform, what developer productivity means to you, the operational conditions you need, and how different database solutions support those goals. We hope this gets you started with understanding the differences between traditional database solutions and Riak, and how you can be successful in a non-relational world.

If you are interested in hearing more about Riak users and use cases, please feel free to browse our resources.

If you have additional questions, get in touch – the Basho team would be happy to discuss your use case and help determine if Riak is the appropriate technological fit.

ABOUT BASHO

Basho Technologies is the leader in highly-available, distributed database technologies used to power scalable, data-intensive Web, mobile, and e-commerce applications and large cloud computing platforms. Basho customers, including fast-growing Web businesses and large Fortune 500 enterprises, use Riak™ to implement content delivery platforms and global session stores, to aggregate large amounts of data for logging, search, and analytics, to manage, store and stream unstructured data, and to build scalable cloud computing platforms. Riak is available open source for download. Riak Enterprise is available with advanced replication, services and 24/7 support. Riak CS enables multi-tenant object storage with advanced reporting and an Amazon S3 compatible API. For more information visit http://www.basho.com.