CAP Theorem

CAP is an abbreviation for consistency, availability, and partition tolerance. The basic idea is that in a distributed system, you can have only two of these properties, but not all three at once. Let's look at what each property means.

- **Consistency**
  Data access in a distributed database is considered to be consistent when an update written on one node is immediately available on another node. Traditional ways to achieve this in relational database systems are distributed transactions. A write operation is only successful when it's written to a master and at least one slave, or even all nodes in the system. Every subsequent read on any node will always return the data written by the update on all nodes.

- **Availability**
  The system guarantees availability for requests even though one or more nodes are down. For any database with just one node, this is impossible to achieve. Even when you add slaves to one master database, there's still the risk of unavailability when the master goes down. The system can still return data for reads, but can't accept writes until the master comes back up. To achieve availability data in a cluster must be replicated to a number of nodes, and every node must be ready to claim master status at any time, with the cluster automatically rebalancing the data set.

- **Partition Tolerance**
  Nodes can be physically separated from each other at any given point and for any length of time. The time they're not able to reach each other, due to routing problems, network interface troubles, or firewall issues, is called a network partition. During the partition, all nodes should still be able to serve both read and write requests. Ideally the system automatically reconciles updates as soon as every node can reach every other node again.

Given features like distributed transactions it's easy to describe consistency as the prime property of relational databases. Think about it though, in a master-slave setup data is usually replicated down to slaves in a lazy manner. Unless your database supports it (like the semi-synchronous replication in MySQL 5.5) and you enable it explicitly, there's no guarantee that a write to the master will be immediately visible on a slave. It can take crucial milliseconds for the data to show up, and your application needs to be able to handle that. Unless of course, you've chosen to ignore the potential

inconsistency, which is fair enough, I'm certainly guilty of having done that myself in the past.

While Brewer's original description of CAP was more of a conjecture, by now it's accepted and proven that a distributed database system can only allow for two of the three properties. For example, it's considered impossible for a database system to offer both full consistency and 100% availability at the same time, there will always be trade-offs involved. That is, until someone finds the universal cure against network partitions, network latency, and all the other problems computers and networks face.

The CAP Theorem is Not Absolute

While consistency and availability certainly aren't particularly friendly with each other, they should be considered tuning knobs instead of binary switches. You can have some of one and some of the other. This approach has been adopted by quorum-based, distributed databases.

A quorum is the minimum number of parties that need to be successfully involved in an operation for it to be considered successful as a whole. In real life it can be compared to votes to make decisions in a democracy, only applied to distributed systems. By distributed systems I'm referring to systems that use more than one computer, a node, to get a job done. A job can be many things, but in our case we're dealing with storing a piece of data.

Every node in a cluster gets a vote, and the number of required votes can be specified for the system as a whole, and for every operation separately. If the latter isn't specified, a sensible default is chosen based on a configured consensus, a path that oftentimes is not successfully applied to a democracy.

In the world of quorum database systems, every piece of data is replicated to a number of nodes in a cluster. This number is specified using a value called N. It represents a default for the whole cluster, and can be tuned for every read and write operation.

Consider a cluster with five nodes and an N value of 3. The N value is the number of replicas, and you can tune every operation with a quorum, which determines the number of nodes that are required for that operation to be successful.

What is Riak?

Riak does one thing, and one thing really well: it ensures data availability in the face of system or network failure, even when it has only the slightest chance to still serve a piece of data available to it, even though parts of the whole dataset might be missing temporarily.

At the very core, Riak is an implementation of Amazon's Dynamo, made by the smart folks from Basho. The basic way to store data is by specifying a key and a value for it. Simple as that. A Riak cluster can scale in a linear and predictable fashion, because adding more nodes increases capacity thanks to consistent hashing and replication. Throw on top the whole shebang of fault tolerance, no special nodes, and boom, there's Riak.

A value stored with a key can be anything, Riak is pretty agnostic, but you're well advised to provide a proper content type for what you're storing. To no-one's surprise, for any reasonably structured data, using JSON is recommended.

Riak: Dynamo, And Then Some

There's more to Riak than meets the eye though. Over time, the folks at Basho added some neat features on top. One of the first things they added was the ability to have links between objects stored in Riak, to have a simpler way to navigate an association graph without having to know all the keys involved.

Another noteworthy feature is MapReduce, which has traditionally been the preferred way to query data in Riak, based for example, on the attributes of an object. Riak utilizes JavaScript, though if you're feeling adventurous you can also use Erlang to write MapReduce functions. As a means of indexing and querying data, Riak offers full-text search and secondary indexes.

There are two ways I'm referring to Riak. Usually when I say Riak, I'm talking about the system as a whole. But when I mention Riak KV, I'm talking about Riak the key-value store (the original Riak if you will). Riak's feature set has grown beyond just storing keys and values. We're looking at the basic feature set of Riak KV first, and then we'll look at things that were added over time, such as MapReduce, full-text search, and secondary indexes.

MapReduce Basics

A MapReduce query consists of an arbitrary number of phases, each feeding data into the next. The first part is usually specifying an input, which can be an entire bucket or a number of keys. You can choose to walk links from the objects returned from that phase too, and use the results as the basis for a MapReduce request.

Following that can be any number of map phases, which will usually do any kind of transformation of the data fed into them from buckets, link walks or a previous map phase. A map phase will usually fetch attributes of interest and transform them into a format that is either interesting to the user, or that will be used and aggregated by a following reduce phase.

It can also transform these attributes into something else, like only fetch the year and month from a stored date/time attribute. A map phase is called for every object returned by the previous phase, and is expected to return a list of items, even if it contains only one. If a map phase is supposed to be chained with a subsequent map phase, it's expected to return a list of bucket and key pairs.

Finally, any number of reduce phases can aggregate the data handed to them by the map phases in any way, sort the results, group by an attribute, or calculate maximum and minimum values.

Mapping Tweet Attributes

Now it's time to sprinkle some MapReduce on our tweet collection. Let's start by running a simple map function. A MapReduce request sent to Riak using the HTTP API is nothing more than a JSON document specifying the inputs and the phases to be executed. For JavaScript functions, you can simply include their stringified source in the document, which makes it a bit tedious to work with. But as you'll see in a moment, riak-js handles this much more JavaScript-like.

Let's build a map function first. Say, we're interested in tweets that contain the word "love", because let's be honest, everyone loves Justin Bieber. `Riak.mapValuesJson()`, used in the code snippet below, is a built-in function that extracts and parses the value of serialized JSON object into JavaScript objects.

```
var loveTweets = function(value) {
  try {
    var doc = Riak.mapValuesJson(value)[0];
```

```
    if (doc.tweet.match(/love/i)) {
      return [doc];
    } else {
      return [];
    }
  } catch (error) {
    return [];
  }
}
```

Before we the look at the raw JSON that's sent to Riak, let's run this in the Node console, feeding it all the tweets in the `tweets` bucket.

```
riak.add('tweets').map(loveTweets).run()
```

Imagine a long list of tweets mentioning Justin Bieber scrolling by, or try it out yourself. The number of tweets you'll get will vary from day to day, but given that so many people are in love with Justin, I don't have the slightest doubt that you'll see a result here.

Using Reduce to Count Tweets

What if we want to count the tweets using the output we got from the map function above? Why, we write a reduce function of course.

Reduce functions will usually get a list of values from the map function, not just one value. So to aggregate the data in that list, you iterate over it and well, reduce it. Thankfully JavaScript has got us covered here. Let's whip out the code real quick.

```
var countTweets = function(values) {
  return [values.reduce(function(total, value) {
    return total + 1;
  }, 0)];
}
```

Looks simple enough, right? We iterate over the list of values using JavaScript's built-in `reduce` function and keep a counter for all the results fed to the function from the map phase.

Now we can run this in our console.

```
riak.add('tweets').map(loveTweets).
                   reduce(countTweets).run()
// Output: [ 8 ]
```

Riak Secondary Indexes

Secondary indexes (or short: 2i) are a very recent addition to Riak. They offer a much simpler way to do reverse lookups on data stored in Riak than Riak Search. Instead of having Riak analyze and tokenize data in documents (JSON, XML, text), 2i relies on the application to provide the indexing data as key-value pairs when storing data. It doesn't do any tokenization either, and doesn't allow things like partial matches like full-text search does, so it's much less computationally expensive.

Computation is instead pushed into the application layer, which basically tags objects stored in Riak with the data it wants to run queries on. Riak 2i can index integers and binary values like strings, and everything is stored in lowercase. So instead of indexing an object you add some metadata to it which Riak can then use to do index lookups. Like Riak Search, 2i only returns keys for you to use. It doesn't do any object lookups, but you can feed the results into MapReduce to do so in the same step.

Querying is kept rather simple too, allowing only full matches and ranges. Don't worry, the folks at Basho are continuously improving on that front, but it's a good start for a more lightweight alternative to Riak Search.

To use Riak 2i, you have to enable the LevelDB storage backend. You can find details on how to do that in the section on storage backends.

Indexing Data with 2i

Index data is stored alongside the objects they're associated with, much like the metadata (links and such) mentioned earlier. Just like metadata, you provide indexing data as additional headers, prefixed with `X-Riak-Index`. Note that the header names are case-insensitive, so any case of `X-Riak-Index` will do.

We'll start by indexing a tweet's username, working off the initial indexing examples in this chapter, in the end adapting our Twitter indexer to use secondary indexes.

Indexes need to be typed, and this is done by adding a suffix to the name identifying the type, currently `_bin` and `_int` are supported. Indexing a string field means it's binary for Riak 2i, so both the username and the date get a `_bin` field name suffix. They end up being sent to Riak as `X-Riak-Index-username_bin` and `X-Riak-Index-tweeted_at_bin` respectively.

riak-js comes with some preliminary support for 2i, but it's more than good enough for our purposes. Secondary indexes are just really simple to build and use. Just add a new `index` attribute to the meta data.

```
tweet = {
  user: 'roidrage',
  tweet: 'Using @riakjs for the examples in the Riak chapter!',
  tweeted_at: new Date(2011, 1, 26, 8, 0).toISOString()
}

riak.save('tweets', '41399579391950848', tweet, {index: {
  username: 'roidrage',
  tweeted_at: new Date(2011, 1, 26, 8, 0).toISOString()
}})
```

The only change we've done is to add some metadata for indexes. riak-js will automatically resolve the field names to have the proper datatype suffixes, so the code looks a bit cleaner than the underlying HTTP request, which we'll look at anyway.

```
$ curl -X PUT localhost:8098/riak/tweets/41399579391950848 \
  -H 'Content-Type: application/json' \
  -H 'X-Riak-Index-username_bin: roidrage' \
  -H 'X-Riak-Index-tweeted_at_bin: 2011-02-26T08:00:00.000Z' \
  -d @-
{
  "username":"roidrage",
  "tweet":"Using @riakjs for the examples in the Riak chapter!",
  "tweeted_at":"2011-02-26T08:00:00.000Z"
}
```

There are special field names at your disposal too, namely the field `$key`, which automatically indexes the key of the Riak object. Saves you the trouble of specifying it twice. Riak automatically indexes the key as a binary field for your convenience, so be sure to avoid using the field `$key` elsewhere. It's also worth mentioning that the `$key` index is always at your disposal, whether you index other things for objects or not. That gives you a nice advantage over key filters when you query Riak for ranges of keys.

That's pretty much all you need to know to start indexing data. There's no precondition, just go for it. It really is the simplest way to get started building a query system around data stored in Riak.

Using Pre- and Post-Commit Hooks

There are scenarios where you want to run something before or after writing data to Riak. It could be as simple as validating data written, for instance to check if the JSON conforms to a well-known schema, or if it's written in the expected serialization format. If the validation fails the code can fail the write, returning an error to the client. The code could also modify the object before it's written, for example to add timestamps or to add audit information.

Another use case we already came across with Riak Search is to update data in a secondary data source with the data just written. Riak Search updates its search index before the data is written to Riak, failing the write if indexing caused an error. That way your application knows right away if there are problems with either the data or your Riak setup.

This feature is called a pre-commit hook, and it's run before Riak sends the object out to the replicas, allowing the hook to control if the write should succeed or if it should fail. A pre-commit hook can be written in JavaScript or Erlang.

A post-commit hook, on the other hand, is run after the write operation is done, and the node that coordinates the request has already sent the reply to the client. What you do in a post-commit hook has no effect on the request as a whole. You can modify the data but you'd have to explicitly write it back to Riak again. Post-commit hooks can be used to update external data sources, for example a search index, to trigger notifications in a messaging system, or to add metrics about the data to a system like Graphite, Ganglia, or Munin. Post-commit hooks can only be written in Erlang.

Let's walk through some examples.

Validating Data

The simplest thing that could possibly work is a JavaScript function that checks if the data written is valid JSON. To validate, the function tries to parse the object from JSON into a JavaScript structure. Should parsing the object fail, the function returns a hash with the key `fail` and a message to the client. Alternatively, the function could just return the string `"fail"` to fail the write.

If parsing succeeds, it returns the unmodified object. To make the code easier to deploy later, it's wrapped into a `Precommit` namespace and assigned to a function variable `validateJson`, so we can call the method as `Precommit.validateJson(object)`.

```
var Precommit = {
  validateJson: function(object) {
    var value = object.values[0].data;
    try {
      JSON.parse(value);
      return object;
    } catch(error) {
      return {"fail": "Parsing the object failed: " + error};
    }
  }
}
```

There is a problem with this code. Pre-commit hooks are not just called for writes and updates, they're also called for delete operations. When a client deletes object, the pre-commit hook will waste precious time trying to decode the object. Riak sets the header `X-Riak-Deleted` on the object's metadata when it's being deleted.

To work around this particular case, we'll extend the code to exit early and return the object when the header is set.

```
Precommit = {
  validateJson: function(object) {
    var value = object.values[0];
    if (value['metadata']['X-Riak-Deleted']) {
      return object;
    }

    try {
      JSON.parse(value.data);
      return object;
    } catch(error) {
      return {"fail": "Parsing the object failed: " + error}
    }
  }
}
```

Unlike MapReduce, JavaScript code for pre-commit hooks needs to be deployed on the Riak nodes. Further down below you'll find a section dedicated to deploying custom JavaScript.

Enabling Pre-Commit Hooks

Given you've deployed the code, we can now tell Riak to use the pre-commit function for a bucket. Like so many other settings in Riak, commit

have a different lineage. You'll just keep creating more and more siblings if they're not reconciled.

Sibling explosion can have the consequence of increased read latency. For every read to the object, Riak has to load more and more data to fetch all the siblings. One piece of data only 10 KB in size is no big deal, but a hundred of it suddenly turn the whole object into 1 MB of data. If all you do is write smaller pieces of data, increased read latency is a good (though only one) indicator that your code creates too many siblings.

Building a Timeline with Riak

Now, you may remember that we wanted to build a timeline, for the tweets we're collecting. Now that we went through the details of modeling data structures for Riak, we have all the information we need to get started. The timeline is nothing more than a list of changes. Every tweet in it is atomic, there are no duplicates, and duplicate writes of the same tweet are easy to filter out thanks to their identifier.

This idea has been made popular by Yammer. They built a notification service on top of Riak that follows a similar way of modeling the data. In fact, they led the way of how to build time series data structures on top of Riak. Hat tip to you once again, Coda Hale. You should make sure to watch their talk at a Riak meetup.

A timeline keeps a list of unique items for a single user, sorted by time. For our purposes, it represents a Twitter user's timeline, based on the tweets that matched the search. The timeline is stored in a Riak object per user and keeps the whole timeline as a list, referencing the tweets. Here's an example.

```
{
  "entries": [
    "1231458592827",
    "1203121288821",
    "1192111486023",
    "1171436045885"
  ]
}
```

The simplest timeline that could possibly work. To make it more efficient we could make it include the entire tweet. Here's a slightly more complex version.

```
{
  "entries": [{
    "id": "1231458592827",
    "username": "roidrage",
    "tweet": "Writing is hard."
  }, {
    "id": "1203121288821",
    "username": "roidrage",
    "tweet": "Finishing up those last chapters."
  }, {
    "id": "1192111486023",
    "username": "roidrage",
    "tweet": "Only two more chapters to go."
  }, {
    "id": "1171436045885",
    "username": "roidrage",
    "tweet": "Almost done with the part on Riak."
  }]
}
```

You can keep adding attributes as you see fit, but it pays to keep the data in the timeline simple. Assuming JSON is the serialization format of choice, every new tweet added to the list adds up to 300 or 400 bytes. With 100 tweets, the Riak object is about 40 KB in size, with 500, it already clocks in at 200 KB. That's not a massive size, but if it keeps growing indefinitely, the Riak object grows bigger and bigger.

Both ways of modeling the timeline share the same advantage. You can assume that the `id` attribute is already respecting time, as that's what Twitter's Snowflake tool does. Snowflake generates unique, incrementing numbers to identify tweets. One part of the generated number is derived from a timestamp. Ordering the entries by that attribute will ensure that they're sorted by time.

Here's the code to handle the timeline, first the part that adds new entries, prepending them to an existing list of entries.

```
var tweet = {
  id: '41399579391950848',
  user: 'roidrage',
  tweet: 'Using riakjs for the examples in the Riak chapter!',
  tweeted_at: new Date(2011, 1, 26, 8, 0)
};

riak.get("timelines", "roidrage",
  function(e, timeline, meta) {
```

```
    if (e && e.notFound) {
      timeline = {entries: []};
    }
    timeline.entries.unshift(tweet.id);
    riak.save("timelines", "roidrage", timeline, meta);
  }
});
```

If no timeline exists, we create a new one and then add the tweet to the beginning of the list. Next up, we'll add the code that reconciles two diverged timelines.

```
function reconcile(objects) {
  var changes = [];
  for (var i in objects) {
    changes.concat(objects[i].data.entries);
  }
  changes.reduce(function(acc, current) {
    if (acc.indexOf(current) == -1) {
      acc.push(current);
    }
    return acc;
  }, []);
  return changes.sort().reverse();
}
```

First, all the changes are collected in one list. The list is then deduplicated, having only single items in it. Lastly, it's sorted and the list reversed, so that the items are in descending order, with newest tweets first.

All that's left to do is update the code saving timeline objects to reconcile potential siblings before storing it back.

```
riak.get("timelines", "roidrage",
  function(e, timeline, meta) {
    if (e && e.notFound) {
      timeline = {entries: []};
    } else if (meta.statusCode == 300) {
      var entries = reconcileConflicts(timeline);
      timeline = timeline[0];
      timeline.entries = entries;
    }
    timeline.entries.unshift(tweet.id);
    if (!meta.vclock) {
      meta = {}
    }
    riak.save("timelines", "roidrage", timeline, meta)
```