Hello.

about

on github

at twitter

via email

home

How You Should Go About Learning NoSQL

2011-08-15 14:01:12 UTC

Yesterday I tweeted three simple rules to learning NoSQL. Today I'd like to expand on that. The rules are:

> 1: Use MongoDB. 2: Take 20 minute to learn Redis 3: Watch this video to understand Dynamo.

Before we get going though, I want to talk about two different concepts which'll help us when we talk about specific technologies.

Secondary Indexes and Joins

First let's talk about *secondary indexes*. In the relational world, you can normally have as many indexes on a table as you want. Although a primary index (or primary key) always has to be unique, there really isn't any appreciable difference between a primary and secondary index (which is any other index which isn't your primary key). We're talking about this because some (though certainly not all!) NoSQL solutions don't offer secondary indexes. Your very first thought might be that this is insanity, but let's just see where it takes us.

Let's imagine that our relational databases didn't have secondary indexes, what would we do? It turns out that managing our own indexes isn't that difficult. Say we have a `Scores` table which had an `Id`, `LeaderboardId`, `UserId` and `Score` column. Our primary index will go on the `Id` column. However, we also want to be get scores based on their `LeaderboardId`. The solution? We create a 2nd table which has two columns: `LeaderboardId` and `ScoreIds`. In this case, we index the `LeaderboardIds` so that we can get all of the `ScoreIds` that belong to a given leaderboard. Whenever we add a new score, we push it onto the `ScoreIds` column. With this list, we can then fetch all the scores by their `Id`. Getting the scores which belong to a leaderboard would be 2 queries (both using an index). First getting all the `ScoreIds` by a given `LeaderboarId`, then getting all the matchin `Score` by `Id`. Storing a score would also be two queries.

Obviously this doesn't work well with relational databases since we'd probably have to treat `ScoreIds` as a comma-delimited string. However, if the storage engine treated arrays as first class objects (so that we can push, remove and slice in constant time) it wouldn't be the most ridiculous approach in the world (although, there's no denying that a secondary index is better).

The other thing we need to talk about are joins. While some NoSQL solutions support secondary indexes and some don't, they almost all agree that joins suck. Why? Because joins and sharding don't really work together. Sharding is the way that most NoSQL solutions scale. Keeping things simple, if we were to shard our above `Scores` example, all the scores for leaderboard 1, 3, 5, 7 and 9 might be on server 1, while server 2 contained all the scores for leaderboard 2, 4, 6, 8 and 10. Once you start to split your data around like this, joining just doesn't make sense. How do we grab the UserName (joined on `Scores.UserId` to `Users.Id`) when users are shared across different leaderboards ?

So, how do we deal with a joinless world? First, NoSQL folk aren't afraid to denormalize. So, the simplest solution to our above problem is to simply stick the UserName within `Scores`. That won't always work though. The solution is to join within your application. First you grab all the scores, from these you extract the `UserIds` and then issue a 2nd query to get the `UserNames`. You are essentially adding complexity in your code so that you can scale horizontally (aka, on the cheap).

MongoDB

With the above out of the way, we can talk about MongoDB. This is easily the first NoSQL solution you should use for a couple of reasons. First, it's easy to get setup on any operating system. Goto this horrible download page (which could make Barry Schwartz write another book), download the right package, unzip, create c:/data/db (or /data/db), startup bin/mongod and you're done. You can connect by either running bin/mongo, or downloading a driver for your favorite programming language.

The other nice thing about MongoDB is that it fully supports secondary indexes and is, aside from the lack of joins, not that different in terms of data modeling. The whole thing is pretty effortless, from setup to maintenance, from modeling to querying. It's also one of the more popular NoSQL solutions, so it's a relatively safe bet. A lot of NoSQL solutions are about solving specific problems. MongoDB is a general solution which can (and probably should) be used in 90% of the cases that you currently use an RDBMS.

MongoDB isn't perfect though. First, the website and online documentation are brutal. Thankfully, the official Google Group is very active and I wrote a free little ebook to help you get started. Secondly, once your working set no longer fits in memory, MongoDB seems to perform worse than relational databases (otherwise it's much faster). As a mixed blessing, MongoDB relies on MapReduce for analytics. It's much more powerful than SQL aggregate capabilities, but it's currently single threaded and doesn't scale like most of us would expect a NoSQL solution to. My final complaint is that, compared to other NoSQL solutions, MongoDB has average availability. It's in the same ballpark as your typical RDBMS setup.

Finally, if you just want to try MongoDB quickly, you can always try out the online tutorial I wrote, you'll be connected to a real MongoDB instance!

Redis

Redis is the most misunderstood NoSQL solution out there. That's a real shame considering you can absolutely 100% master it in about 30 minutes. You can download, install and master Redis in the time it'll take to download SQL Server from MSDN. People (including Redis people) often call Redis a key=>value store. I think the right way to think about Redis is as an in-memory data structure engine. WTF does that mean? It means Redis has 5 built-in data structures which can do a variety of things. It just so happens that simplest data structure is a key value pair (but there are 4 others, and they are awesome).

Now, Redis requires a pretty fundamental shift in how you think of your data. Oftentimes you'll use it to supplement another storage engine (like MongoDB) because some of your data will have been born to sit inside one of Redis data structures while others will be like trying to force a square peg in a round hole. Like MongoDB, Redis is super easy to setup and play with. Windows users will want to use this port for testing. Unlike MongoDB, Redis doesn't support secondary indexes. However, one if its data structure, *Lists*, is perfectly suited for maintaining your own.

Let's look at an example. We keep track of the total number of users and the number of unique (per day) users that log into our system. Tracking the total numbers is easy. We'll use the simplest *String* data structure, which is the key value pair. Our key will be the date, say "2011-08-15" (Redis keys don't have to be strings, any byte data will do). If we visit the *String* documentation we see that they support an `INCR` command. So, for every hit, all we do is `redis.incr(Time.now.utc.strftime('%Y-%m-%d'))`. If we want to get the numbers for the past week, we can do `redis.mget *Array.new(7){|i| (Time.now.utc - (86400 * i)).strftime('%Y-%m-%d') }`.

For our unique users, we'll use the a *Set* structure. On each hit we'll also call `redis.sadd(Time.now.utc.strftime('%Y-%m-%d'), USER)`. We can get the count by using the `scard` command. We don't actually have to worry about duplicates, that's what the Redis' set takes care of for us. (At the end of the day we can turn our set into a simple string value to save space).

Redis isn't a perfect solution though. First, sometimes your data just won't be a good fit. Secondly, it requires that all your data fits into memory (the VM doesn't really work great). Also, until Redis Cluster comes out, you're stuck with replication and manual failover. However, it's fast, well documented and when the model works out (which is often a matter of changing how **you** look at it) you can achieve amazing things with a few lines of code.

I've blogged a bit about Redis Modeling (which I think is the biggest barrier to entry). The first blog post talks about using Redis with MongoDB. The second post looks at dealing with time-based sequences using *Sorted Sets*.

Dynamo/Cassandra

The last point I want to talk about is Cassandra and Dynamo. Dynamo is a set of patterns you can use to build a high-available storage engine. Cassandra is the most popular open source implementation. Now, I know the least about these, so I'm not going to go in any great detail. I will say that you really ought to watch this video which describes Dynamo. The video is given from Riak's point of view, which is another open source Dynamo implementation. But it's pretty generic.

Dynamo is very infrastructure-oriented and might not seem as interesting/relevant to day to day programming. However, the above video is so good (if not a little long), that I think it's well worth it if it makes you think about availability in a new light (as it did for me). Since watching the video I've been pestering the MongoDB folk to implement better availability, it just seems so right.

Where I think you should download and play with MongoDB and Redis today, I think you can take your time around Cassandra or Riak. First, they are both harder to setup. Secondly, they both require changes to how you model your data (in a way that I think is more pervasive than Redis in that you'll probably only use Redis in specific, well-fitting, situations). Finally, and this might just be ignorance on my part, but I found the Ruby cassandra driver to be an absolute nightmare. Java folk will probably have a better time (since Cassandra is written in Java).

In other words, I think Dynamo is worth familiarizing yourself with because I think availability is important, but if you need to use a dynamo-solution, you'll know it (and won't need me to tell you).

Conclusion

There isn't much more to say other than you should just go ahead and have fun. NoSQL is a big world, and solutions vary in complexity and differentness. That's why I think MongoDB, which isn't very different, and Redis, which is different but very simple, are a great place to start.

« Someone is selling my Foundations of Programming ebook on Amazon.com (and it isn't me)

DISQUS

blog comments powered by **DISQUS**