Pairing NoSQL and Relational Data Storage: MySQL with MongoDB

by Max Indelicato on 8/5/2010 9:01:04 PM

Tags: mongodb, amazon ec2, production, database, nosql

I've largely steered clear of publicly commenting on the "NoSQL vs. Relational" conflict. Keeping in mind that this argument is more about currently available solutions and the features their developers have chosen to build in, I'd like to dig into this and provide a decidedly neutral viewpoint. In fact, by erring on the side of caution, I've inadvertently given myself plenty of time to consider the pros and cons of both data storage approaches, and although my mind was initially swaying toward the NoSQL camp, I can say with a fair amount of certainty, that I've found a good compromise.

As developers working in a business context, it is typically our responsibility to ensure that we are taking advantage of all the tools in our toolbelt such that we can deliver on our business requirements as efficiently and effectively as we know how. That said, I propose the following: NoSQL and Relational data stores are not at odds with one another. They are, in my opinion, highly complementary.

First, it's important to realize that thanks to a number of programmatic design patterns – for example, the Repository pattern – we are able to combine data storage methods in an application in a fairly abstract manner. This is HUGE and is the basis from which the rest of my argument is built upon – we are not restricted to any one single implementation of storing data within a contiguous application environment. One part of an application can use Relational storage while another part can use NoSQL storage.

Use Case: A Pay-for-use Analytics Suite

The above theory is sound on the surface, but let's get experiential and apply it to a use case that we might see in the typical planet-scale web development world we now live in. Let's pretend we're developing an Analytics Suite, similar in concept to a pay-for-use version of Google Analytics.

With this Analytics Suite project in mind, here are the high-level business requirements we'll be working with:

1. Payment Transaction Guarantees: We need to be able to collect credit card payments and store their occurrence in a guaranteed manner.
2. Large Unique Visitor and Pageview Dataset: We need to be able to store a large dataset of unique visitors and pageviews.
3. High Performance Insertions: We need to be able to support a high inserts-per-second rate as we record the millions of unique visitors and pageviews that our customer's websites receive every month.
4. Real-time Reporting: We need to be able to report off of our unique visitor and pageview statistics in real-time.
5. High Availability: Our system needs to store data in a manner that provides a high percentage of uptime (i.e 99.99%)

Implementation

We'll implement our architecture with the intent that we'll be using the right tool for the job, as is appropriate for each component within our system. We can simplify our architecture down to two high-level components: a payment datastore and an analytics datastore.Why these two? Our business is split

into two distinct operations:

1. Recording our unique visitor analytics and pageviews
2. Charging our customers for the recorded analytics

Neither operation is tightly coupled with the other, so it is a natural fit for a divide-and-conquer or shared-nothing strategy. We can now take this a step further and get specific about data storage products. Our requirements seem to fit two product types in particular: Relational data stores and NoSQL data stores as they are both typically implemented.

You might be wondering what the justification is for such a broad sweeping conclusion, but by baring with me a bit longer my hope is that the reasoning will become clear as we progress further through this post.

Now let's take a step back and break our business requirements down, one by one, in an effort to clarify our implementation.

Payment Transaction Guarantees

MySQL does transactions well thanks to InnoDB. It's ACID compliant and has full transactional support.

MongoDB, on the other hand, has neither ACID compliance nor full transactional support. In fact, it has no transactional support.

This is a no-brainer. MySQL is the more appropriate tool for the job of payment transactions. And that works out fine for us, because this is a shared-nothing architecture – payment transactions have little to nothing to do directly with the recording of individual unique visitors or pageviews.

We can implement this at the application level as follows:

1. On a monthly basis, we batch process an aggregate of the past month's unique visitor and pageview count in MongoDB.
2. We then insert into MySQL the appropriate payment data based upon the MongoDB data we harvested and its relation to some sort of tiered payment scheme.

I would take this even further and say that all user, account, and payment data in general should be housed within our MySQL database. Can the other stuff be done in MongoDB? Yes, it can, but there are more logical relations between those three groups of data than there are between them and unique visitors/pageviews. Keeping related data together is what MySQL is good at, so let's take advantage of that here.

This requirement's solution was simple and straightforward, so let's keep moving.

Large Unique Visitor and Pageview Dataset

MySQL is effective in dealing with large datasets. But there are large datasets and then there are large datasets. Datasets in the multiple terabyte range are unwieldy in most MySQL setups. You quickly hit up against the hardware limitations of a single machine (short of a massive RAID setup) and are required to look elsewhere for a storage solution. Network Attached Storage (NAS) is a common solution, as is sharding a MySQL dataset across multiple servers.

NAS' tend to be on the expensive side, which is fine if you have an enterprise-level budget. Big hardware is a luxury that is great if you can afford it.

The later alternative, sharding MySQL, is less than ideal for a variety of reasons, one of which is that you need to bring sharding into the application layer. It's doable (and even kind of fun), but we're looking for a solution that "just works" because we're in business and time is money ;)

MongoDB's built-in sharding feature is more in line with what we're looking for in a data storage solution. It allows us to more easily split data up amongst many servers in a transparent way, thereby requiring minimal maintenance and keeping horizontal scalability at the data storage level out of our application layer.

MongoDB is better suited for this requirement than MySQL.
High Performance Insertions

MySQL is a high performance data store, no doubt. But when you get to a point where you need to go from what can be done on a single server, to needing multiple servers to distribute the dataset, you begin to lose some of the advantages of a relational data store as they are implemented in today's relational products.

If we're expecting to receive tens of millions of insertions per month, then it's time to consider some data storage alternatives.

MongoDB fits our needs fairly well in regards to this requirement. It does insertions very fast. Even without citing benchmarks, I feel confident in stating MongoDB does insertions faster than most (if not all) relational databases currently available - and that includes MySQL.

With MongoDB, and its sharding feature, we can easily distribute insertions throughout a series of commodity servers with little to no maintenance or application design overhead. There's no need for us to shard MySQL "manually" at the application level; MongoDB does all that for us – awesome.
Real-time Unique Visitor and Pageview Reporting

Given the sheer quantity of records we're working with, real-time reporting is a tough hurdle to clear. The obvious method may be to simply shard our data over enough servers such that each server is handling a small enough amount of data for us to report on. And that might work, but there's an alternative that I think works better and takes advantage of some of the features inherent in MongoDB.

Real-time reporting on unique visitors and pageviews is only challenging if you're querying across the entire blown-out dataset. Even with MongoDB's Map/Reduce feature, you're still going to have a tough time making reporting fast enough to call it real-time. So instead, let's aggregate unique visitors and pageviews by time.

For example, we could aggregate unique visitors and pageviews by hour, day, and month. After each unique visitor and pageview is recorded, we increment the total count for the current hour, day, and month.

Using the above method may seem like an awful lot of update contention just waiting to happen. However, MongoDB has an "$inc" command that allows us to increment a value atomically. It also has

the additional benefit of being extremely fast thanks to the generally low overhead of insertions in MongoDB.

Aggregating data in this way allows us to report on it quickly and easily. The data is already grouped for us so we're only running a simple retrieval query based on a time clause to get the small number of records required for displaying reports. Instead of querying over tens of millions of records, we're querying over hundreds or thousands – excellent.

MongoDB is clearly the better tool for this requirement at the scale we're working with.

High Availability

Applying high availability to our analytics data store (MongoDB) is accomplished in a manner different than high availability for our payment data store (MySQL) – and rightfully so! MySQL provides high availability primarily via replication in a master-to-master configuration. Is this effective? Yes. Is MySQL's replication feature painless to recover from during a failure? Not quite.

That said – specifically, why is it a good match for our payment feature?

1. We don't have high IO requirements for our payment feature, because it is unlikely that we will have as many customers being charged for usage to the degree that it will max-out MySQL's replication performance.
2. If we experience a failure of the database, we want to fail back to the latest complete transaction because we're dealing with revenue and we all know that there's not much in life more important to a business than accurate recording of its revenue!

On the flip side, we have the analytics data store. MongoDB, and most NoSQL alternatives, are fairly good at pre-packaging high availability into an easy to setup feature-set. MongoDB in particular, with its sharding and replica sets features, allows you to distribute fault across a series of nodes.

It's worth reminding ourselves that MongoDB is not ACID compliant and does not have transactional guarantees. The designers of MongoDB make a point to communicate to users that their solution to high availability is not inherent in transactional guarantees, but in replica sets and distribution of data. Losing a replica or node is "okay" because there should always be more than one in play at any given time allowing you to recover the other replica/node or add a new one as a replacement.

Recap

We've outlined a business use case and it's implementation via two separate but complimentary data storage solutions: MySQL for payment, user, and account information, and MongoDB for unique visitor and pageview analytics data.

I'm making a point here to stay out of the application layer and really provide the bulk of the implementation at the data storage level. In doing so, I've made the assumption that the audience reading this post is familiar enough with web development to fill in the blanks that I've left in regards to the application's code. Hopefully, in doing so, this post still successfully communicates how Relational databases and NoSQL data stores can work in a complimentary way within a single application.

Alternative Implementations and Closing Thoughts

Is this the only combination of data storage technologies that would satisfy our business requirements?

Of course not. I do however think this can be considered one of a very few optimal solutions to a problem domain that many have seen or will see as large-scale development becomes more commonplace.

UPDATE: Interested in interfacing with MongoDB from ASP.NET MVC 2.0? Head on over to our sister site and check out this post here.

Disagree? Love it? Something still unclear? Let me know in the comments below!
About the Author
Max Indelicato

Director of Infrastructure and Software Development at Stride & Associates

Max has worked in a variety of companies, including startups, growth-stage businesses, and established enterprises. He's held the roles of Chief Software Architect, Director of Technology, and the like, where he's built and maintained mobile marketing platforms, large scale public-facing e-commerce websites, and a series of financial applications supporting fixed-income securities financial entities.

Follow Me
. . .
Comments
Posted by Jad Y on 8/5/2010 11:34:32 PM

Nice article, but it seems to me a little inaccurate when talking about pageviews and unique visitors. mongo (and similar nosql stores) can do a great job collecting pageviews, since pageviews are perfectly sequential, therefore the map-reduce task will be straight forward. However not the same applies for unique visitors; If a node is running a task on a shard -say for logs recorded on day X- calculating the number of unique visitors on that segment varies depending on the visitors who came on day X-1 and so on..
Posted by Max Indelicato on 8/6/2010 12:37:07 AM

Jad, good point about the Unique Visitors and thanks for commenting. They would be trickier to get right, but is still possible given that you could MapReduce on all visitors within the current day and aggregate those into hourly, daily and monthly segments. I'm also imagining that an Upsert might be viable here as well - only inserting a new unique visitor if they don't exist during a certain time frame, otherwise a visitor's record is updated to indicate they have just returned.

Pageviews are straightforward - one record for ever pageview sequentially based on time. MapReduce will be effective here, but even MapReduce falls down on datasets that are large enough (10's of millions of records). In theory though, you're right. You could just horizontally scale your nodes to a number that mitigates this issue. Costly, but doable.
Posted by Casey on 8/6/2010 3:16:07 PM

Have you considered using something like InfoBright for analytics? It's a MySQL based data warehousing product that uses a column-oriented architecture for storing data. I did a very small benchmark and it took a CSV file of I think 200MB down to less than 50MB of storage in the DB. Also, the nasty query I ran to test the speed of the database ran twice as fast on a dinky little VM vs. a big Oracle instance. The fact that it can compress the data so well and that it's fast may make it a better solution for analytics, especially if you need ad-hoc reporting and you are loading data in batches.

Also, you can use Pentaho or similar as a reporting front-end.

Granted, this means you'll have three technologies for data stores, but if you want the best tool for the job, this may be the way to go. If nothing else, it's another tool to keep in mind for future efforts.
Posted by Max Indelicato on 8/6/2010 7:15:43 PM

Casey, you're right, columnar data stores are a good alternative to the traditional row-based SQL storage solution in the context of Analytics storage. I haven't spent much time with columnar warehousing, I'll have to check InfoBright's solution out.

To your point about having three data stores, I don't think that's a bad thing if each data store's strengths are being put to good use. It's tougher to maintain, but the trade-off could be worth it in the right scenario.