STARTUP LESSONS LEARNED
by Eric Ries

Sunday, January 4, 2009

Sharding for startups

The most important aspect of a scalable web architecture is data partitioning. Most components in a modern data center are completely stateless, meaning they just do batches of work that is handed to them, but don't store any data long-term. This is true of most web application servers, caches like memcached, and all of the network infrastructure that connects them. Data storage is becoming a specialized function, delegated most often to relational databases. This makes sense, because stateless servers are easiest to scale - you just keep adding more. Since they don't store anything, failures are easy to handle too - just take it out of rotation.

Stateful servers require more careful attention. If you are storing all of your data in a relational database, and the load on that database exceeds its capacity, there is no automatic solution that allows you to simply add more hardware and scale up. (One day, there will be, but that's for another post). In the meantime, most websites are building their own scalable clusters using sharding.

The idea behind sharding is simple: split the data between multiple machines, and have a way to make sure that you always access data from the right place. For example, consider this simple scheme. Imagine you want to store data about customers, each of whom has "last name" field in your database. You could create 26 identical databases, and assign each of them a letter of the alphabet. Then, whenever you want to look up data about John Smith, you first connect to the "S" database, and then fetch the data you want. All of the sudden, your single database solution just got twenty-six times more capacity added to it.

Devotees of sharding will already be groaning, because this is a bad idea. The problem should be familiar to anyone who plays Scrabble: not all letter are equally likely to be used as the first letter of someone's name. The shards for S and R will be heavily loaded, but the shards for Q, X, and Z will probably be idle.

A big part of sharding is picking a good partitioning scheme. Many have been proposed, and I won't discuss them in detail here. I really enjoy reading stories of how people have tackled this problem. Two of my favorites are from Flixster and LiveJournal. In the end, they fall into three broad categories:

1. Vertcal partitioning. In this scheme, all of the data related to a specific feature of a product are stored on the same machines. For example, Friendster was famously vertically partitioned at one time in its growth curve. A different cluster of machines served each of Profiles, Messages, Testimonials, etc.

2. Key-based partitioning. In this scheme, you use part of the data itself to do the partitioning. The "letters of the alphabet" scheme I presented earlier is one (bad) example of this type. More common is to use a one-way hashing algorithm to map the data to be accessed to one of the shards that store it. Certain data types present natural hashes as well: for numeric keys, you can take the key mod N, where N is the number of shards; dates are easily partitioned by time interval, etc.

3. Directory-based partitioning. This scheme maintains a lookup table somewhere in the

cluster, that simply keeps track of which data is stored on which shard. This has two major drawbacks: the central directory can become a single point of failure, and there is a performance cost for having to consult the directory ever time you want to access data anywhere in the cluster.

So far, this is just a summary of what all of us who have attempted to build web-scale architectures considers obvious. For a good in-depth look at each partitioning scheme, I recommend Scalability Strategies Primer: Database Sharding.

Sharding for startups

To support a single partitioning scheme is easy, especially if you design for it from the start. But startups rarely have either luxury. It's almost impossible to know ahead of time what kind of data partition makes the most sense for any particular part of your application. It also doesn't make sense to design your app for scalability from the start, since you may have to create many iterations before you find a version that requires any scalability at all. Even worse, it's a rare app that requires only one kind of data partitioning.

The solution is to build an architecture that works for the startup condition. To me, the optimal architecture is one that serves four goals simultaneously. I'll talk about those goals, and then propose a system that I believe serves them well.

1. The ability to handle increased capacity incrementally, by adding proportional amounts of new hardware. This is the main reason most people undertake sharding in the first place. Note that most key-based partitioning systems do badly on this criteria. Most key-hashing schemes have some breaking point where, if you add just one more shard, you have to re-hash all of the data (imagine having to add a 27th database server to the alphabet-hashing scheme discussed above).

2. Incremental and proportional software complexity. As you increase the amount of data partitioning, application code gets more complex. Just to cite one common example, when all of your data resides on a single shard, you can run complex joins against it. If that same data is partitioned, you need to process the data in other ways. Just as we want to be able to make proportional investments in hardware to handle increased load, a good architecture allows us to make proportionally-sized changes to the code. We want to avoid anything that requires a massive rewrite just to change how data is partitioned.

3. Support multiple sharding schemes. Log-based data is often optimally stored in date-based partitions. Customer data is normally partitioned according to customer id. What about log data about customers? A good architecture allows us to use the right tool for the job, or even combine tools as necessary.

4. Easy to understand. Some sharding schemes are themselves hard to follow, or require you to write code that is hard to read. Especially as teams grow, a good scheme can be undone by programmers who use it wrong. It's unrealistic to expect everyone on the team to understand scalability in detail. Instead, a good architecture has clear rules that allow them to do the right thing without having to think too hard about it.

With those criteria in mind, let me propose a solution that has served me well in several startups. First, I'll describe how it works, and then I'll lay out how to build it incrementally.

A URL-based sharding system

Start by thinking of every kind of data in your application as associated with some entity. Give each entity a name. Then, whenever you want to have your system find the shard for a given entity, compose this name into a URL. For example, let's say you have data about customers. A given customer would have an entity URL that looked like this:

> customer://1234

► May (8)
► April (5)
► March (11)
► February (10)
▼ January (10)
Achieving a failure
Refactoring yourself out of business
Three freemium strategies
Lean hiring tips
Why PHP won
CPI > CPC
Lessons Learned office hours
Sharding for startups
Lessons Learned on Mashable today
Happy new year
► 2008 (59)

Popular Posts

Minimum Viable Product: a guide
One of the most important lean startup techniques is called the minimum viable product . Its power is matched only by the amount of confusi...

What is customer development?
When we build products, we use a methodology. For software, we have many - you can enjoy a nice long list on Wikipedia. But too often when ...

The lean startup
(Update April, 2011: In September, 2008 I wrote the following post in which I published my thoughts on the term " lean startup "...

The Hacker Way
I don't normally comment on the day's news, but I want to make an exception today to share something from Facebook's S-1 filing ...

Recent & Upcoming Events

Plancast
I am experimenting with using Plancast to track my event schedule. Take a look and let me know what you think.

@ericries on Twitter

visit twitter

Venture Hacks interview: "What is the minimum viable product?"

The Principles of Product Development Flow

The Lean Startup Book is here

Top Ten Lies of Entrepreneurs - from Guy Kawasaki

[?]

Manage your startup's Cap Table? try Cap table modeling for startups

Somewhere in your API, you have a way to access data about a given customer. I'm not talking about high-level abstractions (like an object-oriented view of a customer's data). I'm talking about the actual data-fetching operation. At some level in your system, there's probably code that looks something like this:

```
$connection = new_db_connection("host", port, "dbname");
$statement = $connection->prepare($sql_statement, $params);
$result = $statement->execute();
```

What I propose is that you make your sharding decisions directly at the "new_db_connection" seam in the code. The new code would look like this:

```
$connection = new_db_connection("customer://1234");
$statement = $connection->prepare($sql_statement, $params);
$result = $statement->execute();
```

All of the complexity of determining which shard you need to use (which we'll get to in a moment) is hidden behind that simple API. Once a shard-lookup decision is made, there's no additional overhead; your application is talking directly to the right database.

In some cases, you can make it even simpler. If you're of the type of organization that doesn't use db access objects directly, but simply passes SQL statements through a central API, you can try this: consider the entity identifier as part of the statement itself. I've done this before using a structured comment added to each SQL statement, like this:

```
/*entity customer://1234 */ SELECT name FROM customer WHERE id = 1234
```

At the expense of some additional parsing overhead in your data-access layer, you get additional metadata associated with every query in your application. You may start out using this data only for sharding, but you may find it has other benefits as well. For example, you might notice that caching gets a lot easier if you have good metadat about which queries are associated with the same entity. It's a very readable approach, and it makes it easy to change the sharding for a query without having to change a lot of intervening function signatures (since the shard info is carried by the statement that's already passed through).

Whichever API you use, URL-based sharding is very easy to understand. I've taught everyone from hard-core programmers to scripters to HTML-designers to use it properly. And implementing it is easy, too. Every language already has a facility for parsing URLs. I normally also use URLs to specify the shards themselves, too. So, at its heart, the sharding system is really a way of mapping entity URLs to shard URLs, which look like this:

```
customer://1234 -> mysql://hostname:port/dbname
```

The scheme part of the shard URL can allow you to use multiple shard-storage types. For example, in one system I built using this technique, the scheme specified the name of the ADODB driver to use for accessing that database.

No system is entirely without drawbacks, however, and URL sharding is not an exception. In this case, the big drawback is that it works best if you store the mappings in a central directory. This gives you the big payoff of being able to support all kinds of sharding with just one system. Each kind of shard just maps to an appropriate kind of URL:

```
customer://1234 (id)
forums://master (vertical)
log://January/2008 (date)
```

You can even do more complex schemes, like having groups of IDs all map to the same URL in a bucket system, although I recommend keeping it simple whenever possible.

If you have a solid caching system in your cluster, like memcached, the overhead of a system like this is really quite small. The mappings never change (unless you build a data

migration tool, in which case they change rarely) and so are easy to cache. Using memcached's multiget, which allows the fetching of many keys in parallel, I have written code to aggregate all the shard lookups for a given page and prefetch them, reducing the overhead even further.

The actual directory itself is straightforward. You need to store two kinds of information. One is a list of all the available shard types, and the machines available to store those types. This information is normally maintained by your operations team. Whenever they want to brng new capacity online, they simply add a row to the appropriate table. The second is the mapping itself. The sharding API follows a simple rule: whenever asked to find the shard for a URL, it either pulls it from the directory or, if it's not there, assigns it to one of the available shards for that entity type. (There's some added complexity as shards get "weighted" for more or less capacity, and for cases where data needs to be moved between shards, but these are not too serious to deal with). I normally recommend you just store this directory on your master database, but you could use a standalone vertical shard (or even a key-based partition!) to store it.

Incremental build-out

The most important attribute of URL-based sharding, though, is that it easily supports an incremental build-out that is perfect for startups. We can start, as most startups do, with a single, central master DB. We can take advantage of the many open source libraries out there that don't scale especially well, and write as much non-scalable code as we want. When we finally have a product that starts to get traction, we can practice Just-in-Time Scalability, as follows:

1. When load on the master DB gets above some agreed-upon threshold, we measure which tables and queries are causing the most read/write traffic.
2. We pick one such query, and decide to shard it. We always try to find queries that are causng load but are also relatively self-contained. For example, most applications usually have a number of highly-trafficked tables that are rarely joined with any others; these are excellent candidates for early sharding.
3. Once we've picked a query, we have to identify all of the tables it touches, and all of the queries that access those tables. The better your internal API's and data architecture, the easier this step will be. But even for total spaghetti code, this doesn't take more than a few hours.
4. We change the code to have all of the queries identified in step 3 tagged with the apporpriate entity URL that they affect. If we've uncovered an entity type that's not previously been sharded, we'll have to add some new shards and shard-types to the central directory.
5. Since there is already some data about these entities on the master DB, we need to migrate it to the new, sharded, locations we picked in step 4. I won't go into detail about how to do this data migration in this post; coordinating it with the code changes in step 4 is the hard part. You can read a little about it in my post on Just-in-Time Scalability.

Each time we need to add capacity to our application, we go back to step 1. At IMVU, during our steepest growth curves, we'd have to do this exercise every week. But we could do the whole thing in a week, and be ready to do it again the next week. Each time, our application got a little more complex, and a little more scalable. But at not time did we have to take the site down for a rewrite.

If your experience is anything like mine, you may be surprised at how much cruft remains on your master database. One advantage of this system is that it doesn't require you to add scalability to any piece of data that is seldom accessed. All that random junk sitting in your master databsase's scheme may make you feel bad, but it's actually not causing any harm. Only the parts of your app that need to scale will get the sharding treatment.

What do you think?

I've used this setup successfully, and have recommended it to many of the startups I advise. I much prefer it to the more complex solutions I've seen and worked with over the years, even though I recognize it has some drawbacks compared to those other systems. I'd really like to hear your feedback. Go ahead, poke holes in the design. Let me know the use cases that I've overlooked. And most of all, share with all of us what's worked for you.

You might like:

- Don't be the Ice Cream Glove
- Why vanity metrics are dangerous
- Pivot, don't jump to a new vision
- Minimum Viable Product: a guide

Recommended by

21 comments:

Imran January 5, 2009 at 9:36 AM

How does your approach compare to Consistent Hashing?

Reply

Eric January 5, 2009 at 9:49 AM

Consistent hashing has the property that only a small number of entries need to be remapped when a node is added or removed. This is great for something like a huge DHT or a distributed work queue. But with sharding, any remapping is actually pretty painful (since moving data between servers is hard if you want to do it without any data loss, data downtime, and still maintain transactions).

To answer your question directly: 1) URL-based sharding never requires you to remap data, ever and 2) URL-based sharding is a lot simpler to use. There's no complex algorithm to go wrong, just a simple lookup table.

Of course, you could use URL-based sharding to "wrap" a CH algorithm (or any hashing scheme you wanted). And that's the other point I was trying to make in this post: having a consistent data _addressing_ scheme is almost more important that the particular partition scheme you choose.

Does that make sense?

Reply

Nathan January 5, 2009 at 5:37 PM

Thanks for sharing this technique for abstracting sharding policy from the code that queries the sharded data sources. I like it.

I've worked on some systems in the past that had separate sharding policies for different data sources, each with their own data-type-specific function for obtaining a handle to a db connection/rpc stub/etc and the sharding policy decisions were made in that function. Using a more generic obtain-a-handle function that gets passed a URL is nice, as long as the same type of handle can be used for multiple data sources.

For data stored in MySQL it's a no-brainer but it's not quite as obvious how to apply this pattern in systems that use an RPC mechanism such as Thrift. Maybe instead of a new_db_connection("customer://1234") function that returns a db connection

handle it would be more useful in such systems to have a function like addr_for_shard("customer://1234") that would return host & port for the appropriate service share or whatever info is necessary to construct an RPC stub.

Reply

spanky January 5, 2009 at 6:07 PM

E,
How is WoW sharded? I've noticed queues for instances lately...

Reply

dodo January 6, 2009 at 7:19 AM

here is my understanding:
if you shard some tables to a new server,will the sql statements updating or deleting those tables be pointed to the new server? how can we keep a complete view of all the data of ONE system? using replicas? how to prevent lags?

Reply

Raph January 7, 2009 at 12:49 PM

Whoa. "Sharding?" A quick Google seems to show it as a term coming from someone who worked at Flickr... which came of course, from an MMO project (Game Neverending). And in MMOs, of course, it was a fiction-derived term for parallel servers which has managed to stick (many people refer to parallel VW servers as "shards" now).

Did a DB term of art come from UO? How odd.

Reply

Eric January 7, 2009 at 3:07 PM

Raph, as usual you've hit the nail right on the head. I've always liked the term sharding precisely because it has a well-known video game antecedent. That makes it easy for people to visualize. The only difference (ok, one of many, but still) is that WoW shards have way more interesting names. Still, the idea that your character exists on only one shard makes intuitive sense, and it's that sense that is easy to generalize.

Thanks for stopping by,

Eric

Reply

Steve Jenson January 7, 2009 at 9:25 PM

There are two flaws with the idea of storing your shard mappings in memcache. Memcached is not persistent; Memcache keys naturally fall out as new data enters whatever slab your mappings are in and don't forget that machines do restart from time to time.

Reply

Steve Jenson January 7, 2009 at 9:28 PM

I don't think sharding came from the MMO world, we were using the term at Google before MMOs started becoming popular.

Even though I had done it before, I had never heard the term 'sharding' before

joining Google (early 2003) so I always thought the term originated there.

Reply

Raph January 8, 2009 at 11:53 AM

Eric,

I stop by regularly! I just don't comment. :)

Steve,

The term was in active use in 1996 in the MMO world, which predates Google's founding.

I asked some of our DB guys here and they said "bah, another name for vertical partitioning!" :) But they also said "I always wondered where the term came from..."

Reply

Steve Jenson January 8, 2009 at 2:43 PM

Hey, I stand corrected! I always wondered, too.

Reply

Steve Jenson January 8, 2009 at 3:11 PM

A couple of ways I've seen sharding done in the past or have done it myself have been:

consistent hashing: only having to rebalance a small percentage of your data hosts when you add a new database is much preferred over regular hashing where full rebalancing is needed. Nobody I know uses plain hashing for that reason.

If you have a natural partition point, and a natural place to store data, like a blog and a blog's owner, you can store which partition a blog is on with the blog owner's user record. In fact, that's how we did it with Blogger for a long time.

Systems that don't have natural partition points e.g., they need large subsets of data to be available for local JOINS or querying, have a much more difficult time of sharding. You have to take those queries built using JOINS and populate them in separate systems, like Facebook does using Cassandra. Take a look at some of their recent presentations on the subject of using Cassandra to build indices.

Cassandra is ultimately designed to act like Google's BigTable which automatically partitions data according to locality. Similarly to the 'natural partition point' example I gave above, it uses a master (with replica) to keep track of where keys are stored on what tablet servers.

Check out the BigTable paper. I miss using it and am hoping that Cassandra eventually stacks up to BigTable.

Reply

Anonymous January 16, 2009 at 10:52 PM

It seems to be taken as a given in this article that you're going to be using relational databases. Why?

Reply

BillG January 19, 2009 at 9:45 AM

By pushing the shard-selection logic down into the db connection access code, you abstract the fact that the system is sharded from the caller.

This is a convenient approach when you know that ever query for an asset will only go to a single shard, but what if you have to aggregate across multiple shards? Not a JOIN necessarily, but an IN clause or an order by date criteria?

No doubt this requirement will arise just after you've partitioned your data. How would your application code need to be modified to support this since the sharding is abstracted at a low level?

Reply

Anonymous February 23, 2009 at 11:55 PM

Agree with BillG. Nearly every SQL query I see has a join. This type of vertical partitioning 'sharding' scheme won't work in most cases. And, it will lead the developer to write badly normalized databases. However, isolating the data load-balancing decisions in the db abstraction layer makes good sense.

Riffing off of your url idea, what about using a parser in the db class to sniff incoming sql queries (or urls), and connect accordingly to a pair of mysql master/master servers. SELECT's go to the read-optimized server, INSERT's and UPDATE's go to the write-optimized server. Since both servers read and write (and flood data back and forth), there is a lot of flexibility when making improvements to the balancing algorithm. Not to mention built-in data backup (heh).

It would be a short step to spawn more mysql servers and create a cluster, if the load became enormous enough.

Reply

Anonymous March 17, 2009 at 9:06 AM

hi, your artical and information is able to help me. Do you provide any consultancy services on how to design a Shard base architecture? what is your company web site or email address?

Reply

Eric March 17, 2009 at 10:34 AM

Yes, sometimes I do. You can get in touch at startuplessonslearned-eric(at)sneakemail(dot)com or find me on twitter, facebook, or linkedin

Reply

Hot Fuzz March 20, 2009 at 5:26 PM

interesting post

Reply

Small business web site design April 2, 2009 at 11:00 PM

nice post

Reply

web design India April 19, 2009 at 10:49 PM

Your post is helpful and informative

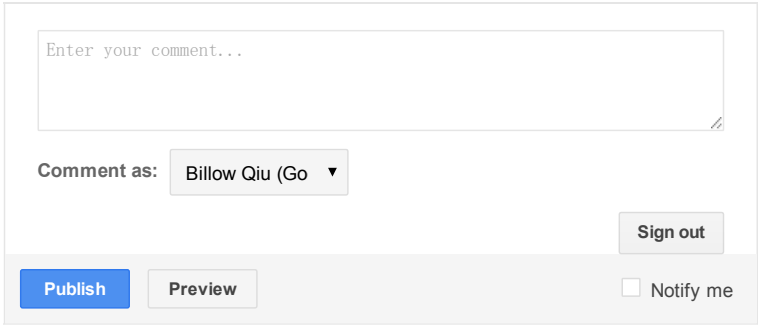
Reply

Jerry December 8, 2009 at 7:16 PM

This is a good summary and very useful data layer scaling approach.

You can easily extend the approach you recommend by adding a web services layer (with appropriate caching) on top of this data access layer and under the covers adding different types of data stores which work ideally for particular types of data (like Tokyo Cabinet for a highly write intensive fixed width key value store).

Reply



Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)