An Unorthodox Approach to Database Design : The Coming of the Shard

Thursday, August 6, 2009 at 3:24PM

Todd Hoff in Example, Shard, Strategy

Update 4: Why you don’t want to shard. by Morgon on the MySQL Performance Blog. *Optimize everything else first, and then if performance still isn’t good enough, it’s time to take a very bitter medicine.*
Update 3: Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes by Dare Obasanjo. Excellent discussion of why and when you would choose a sharding architecture, how to shard, and problems with sharding.
Update 2: Mr. Moore gets to punt on sharding by Alan Rimm-Kaufman of 37signals. Insightful article on design tradeoffs and the evils of premature optimization. With more memory, more CPU, and new tech like SSD, problems can be avoided before more exotic architectures like sharding are needed. Add features not infrastructure. Jeremy Zawodny says he's wrong wrong wrong. we're running multi-core CPUs at slower clock speeds. Moore won't save you.
Update: Dan Pritchett shares some excellent Sharding Lessons: Size Your Shards, Use Math on Shard Counts, Carefully Consider the Spread, Plan for Exceeding Your Shards

Once upon a time we scaled databases by buying ever bigger, faster, and more expensive machines. While this arrangement is great for big iron profit margins, it doesn't work so well for the bank accounts of our heroic system builders who need to scale well past what they can afford to spend on giant database servers. In a extraordinary two article series, Dathan Pattishall, explains his motivation for a revolutionary new database architecture--sharding--that he began thinking about even before he worked at Friendster, and fully implemented at Flickr. Flickr now handles more than 1 billion transactions per day, responding in less then a few seconds and can scale linearly at a low cost.

What is sharding and how has it come to be the answer to large website scaling problems?

Information Sources

- Unorthodox approach to database design Part1:History
- Unorthodox approach to database design Part 2:Friendster

What is sharding?

While working at Auction Watch, Dathan got the idea to solve their scaling problems by creating a database server for a group of users and running those servers on cheap Linux boxes. In this scheme the data for User A is stored on one server and the data for User B is stored on another server. It's a federated model. Groups of 500K users are stored together in what are called *shards*.

The advantages are:

- **High availability**. If one box goes down the others still operate.
- **Faster queries**. Smaller amounts of data in each user group mean faster querying.
- **More write bandwidth**. With no master database serializing writes you can write in parallel which increases your write throughput. Writing is major bottleneck for many websites.
- **You can do more work**. A parallel backend means you can do more work simultaneously. You can handle higher user loads, especially when writing data, because there are parallel paths through your system. You can load balance web servers, which access shards over different network

paths, which are processed by separate CPUs, which use separate caches of RAM and separate disk IO paths to process work. Very few bottlenecks limit your work.

How is sharding different than traditional architectures?

Sharding is different than traditional database architecture in several important ways:

● **Data are denormalized**. Traditionally we normalize data. Data are splayed out into anomaly-less tables and then joined back together again when they need to be used. In sharding the data are denormalized. You store together data that are used together.

This doesn't mean you don't also segregate data by type. You can keep a user's profile data separate from their comments, blogs, email, media, etc, but the user profile data would be stored and retrieved as a whole. This is a very fast approach. You just get a blob and store a blob. No joins are needed and it can be written with one disk write.

● **Data are parallelized across many physical instances**. Historically database servers are scaled up. You buy bigger machines to get more power. With sharding the data are parallelized and you scale by scaling out. Using this approach you can get massively more work done because it can be done in parallel.

● **Data are kept small**. The larger a set of data a server handles the harder it is to cash intelligently because you have such a wide diversity of data being accessed. You need huge gobs of RAM that may not even be enough to cache the data when you need it. By isolating data into smaller shards the data you are accessing is more likely to stay in cache.

Smaller sets of data are also easier to backup, restore, and manage.

- **Data are more highly available**. Since the shards are independent a failure in one doesn't cause a failure in another. And if you make each shard operate at 50% capacity it's much easier to upgrade a shard in place. Keeping multiple data copies within a shard also helps with redundancy and making the data more parallelized so more work can be done on the data. You can also setup a shard to have a master-slave or dual master relationship within the shard to avoid a single point of failure within the shard. If one server goes down the other can take over.

- **It doesn't use replication**. Replicating data from a master server to slave servers is a traditional approach to scaling. Data is written to a master server and then replicated to one or more slave servers. At that point read operations can be handled by the slaves, but all writes happen on the master.

Obviously the master becomes the write bottleneck and a single point of failure. And as load increases the cost of replication increases. Replication costs in CPU, network bandwidth, and disk IO. The slaves fall behind and have stale data. The folks at YouTube had a big problem with replication overhead as they scaled.

Sharding cleanly and elegantly solves the problems with replication.

Some Problems With Sharding

Sharding isn't perfect. It does have a few problems.

- **Rebalancing data**. What happens when a shard outgrows your storage and needs to be split? Let's say some user has a particularly large friends list that blows your storage capacity for the shard. You need to move the user to a different shard.

On some platforms I've worked on this is a killer problem. You had to build out the data center correctly from the start because moving data from shard to shard required a lot of downtime.

Rebalancing has to be built in from the start. Google's shards automatically rebalance. For this to work data references must go through some sort of naming service so they can be relocated. This is what Flickr does. And your references must be invalidateable so the underlying data can be moved while you are using it.

- **Joining data from multiple shards**. To create a complex friends page, or a user profile page, or a thread discussion page, you usually must pull together lots of different data from many different sources. With sharding you can't just issue a query and get back all the data. You have to make individual requests to your data sources, get all the responses, and the build the page. Thankfully, because of caching and fast networks this process is usually fast enough that your page load times can be excellent.

- **How do you partition your data in shards?** What data do you put in which shard? Where do comments go? Should all user data really go together, or just their profile data? Should a user's media, IMs, friends lists, etc go somewhere else? Unfortunately there are no easy answer to these questions.

- **Less leverage**. People have experience with traditional RDBMS tools so there is a lot of help out there. You have books, experts, tool chains, and discussion forums when something goes wrong or you are wondering how to implement a new feature. Eclipse won't have a shard view and you won't find any automated backup and restore programs for your shard. With sharding you are on your own.

- **Implementing shards is not well supported**. Sharding is currently mostly a roll your own approach. LiveJournal makes their tool chain available. Hibernate has a library under development. MySQL has added support for partioning. But in general it's still something you must implement yourself.

See Also

- The Flickr Architecture for more interesting ideas on how to implement sharding.
- The Google Arhitecture.
- The LiveJournal Architecture. They talk quite a bit about their sharding approach and give a lot of helpful details.
- The Shard category.
