percona.com

Why you don't want to shard. - Percona Database Performance Blog

Morgan Tocker

5-6 minutes

Note: *This blog post is part 1 of 4 on building our training workshop.*

The Percona training workshop will *not* cover sharding. If you follow our blog, you'll notice we don't talk much about the subject; in some cases it makes sense, but in many we've seen that it causes architectures to be prematurely complicated.

So let me state it: **You don't want to shard.**

Optimize everything else first, and then if performance still isn't good enough, it's time to take a *very* bitter medicine. The reason you need to shard basically comes down to one of these two reasons:

1. **Very large working set** – The amount of memory you require to keep your frequently accessed data loaded exceeds what you can (economically) fit in a commodity machine. 5 years ago this was 4GB, today it is 128GB or even 256GB.Â Defining "working set" is always an interesting concept here, since with good schema and indexing it normally doesn't need to be the same size as your entire database.
2. **Too many writes** – Either the IO system, or a slave can't keep up with the amount of writes being sent to the server.Â While the IO

system can be improved with a RAID 10 controller w/battery backed write cache, the slave delay problem is actually very hard to solve. Maatkit has a partial-solution (via Paul Tuckfield), but it doesn't work for all workloads.

(Yes, I am simplifying some of the scalability issues with MySQL on big machines, but I have faith that Yasufumi is making this better).

What types of Sharding are there?

Despite my cautions, if you have established that *you need to shard* there are quite a few options available to you:

1. **Sharding Partitioning by Application Function** – This is usually the **best** way to fix any of the problems mentioned above. What you do is pick a few very busy tables, and move them onto their own MySQL server.Â Partition-by-function keeps the architecture still simple, and should work for most cases unless you have a single table which by *itself* can't fit into the above constraints.
2. **Sharding by hash or key** – This method works by picking a column on a table and try and divide up your data based on it.Â You can choose any column to hash on, you just need to make sure that it will equally distribute the data equally. In practice this method can be really hard to get working right, since even if each shard has the same amount of 'customers', demanding users tend to by far exceed average users and some servers are overloaded while others are not.
   (Tip: There are a few famous cases of both (a) bad hashing algorithms and (b) users becoming unequal all of the sudden;Â You don't want to shard based on the first character of a username – as there will be a lot more 'M' than 'Z'.Â For users becoming unequal all of the sudden, it's always interesting to think of what scaling challenges Flickr would have had for the official Obama photographer in the lead up to the 08 election.)

3. **Sharding via a Lookup Service** – This method works by having some sort of directory service which you query first to ask "what shard number will this users data exist on?".Â It's a highly scalable architecture, and once you write scripts to be able to migrate users to/from shards you can tweak and rebalanced to make sure that all your hardware is utilized efficiently.Â The only problem with this method is what I stated at the start: *it's complicated*.

   *(**Note:** I've left out some of the more complicated sharding architectures.Â For example; another solution is to have shards all store fragments of data, and to cross backup those fragments across shards.)*

   Why is it so complex?

   The reason it's complex comes down to two reasons:

1. The application developer has to write more code to be able to handle sharding logic (this is actually lessened with projects such as HiveDB.)
2. Operational issues become more difficult (backing up, adding indexes, changing schema).

   I think that a lot of people remember (1), but (2) can be a real pain point.Â It can take a lot of work to build an application that works correctly when you are rolling through an upgrade where the schema will not be the same on all nodes.Â A lot of these tasks remain only semi-automated, so from an operations perspective there can often be a lot more work to be done.

   This concludes Part 1 – I hope I've justified why we are not covering sharding.Â In Part 2, I will write about something that is going to be in the course – "XtraDB: The top 10 enhancements", and in Part 3 "XtraDB: The top 10 parameters".