Sharding: In Theory and Practice (Part Two)

Standard / by Neil Harkins (https://www.clustrix.com/author/neil-harkins/) / January 17, 2013 / No Comments (https://www.clustrix.com/bettersql/sharding-theory-algorithmic-sharding/#respond)

Part Two: The Differences Between Algorithmic and Dynamic Sharding

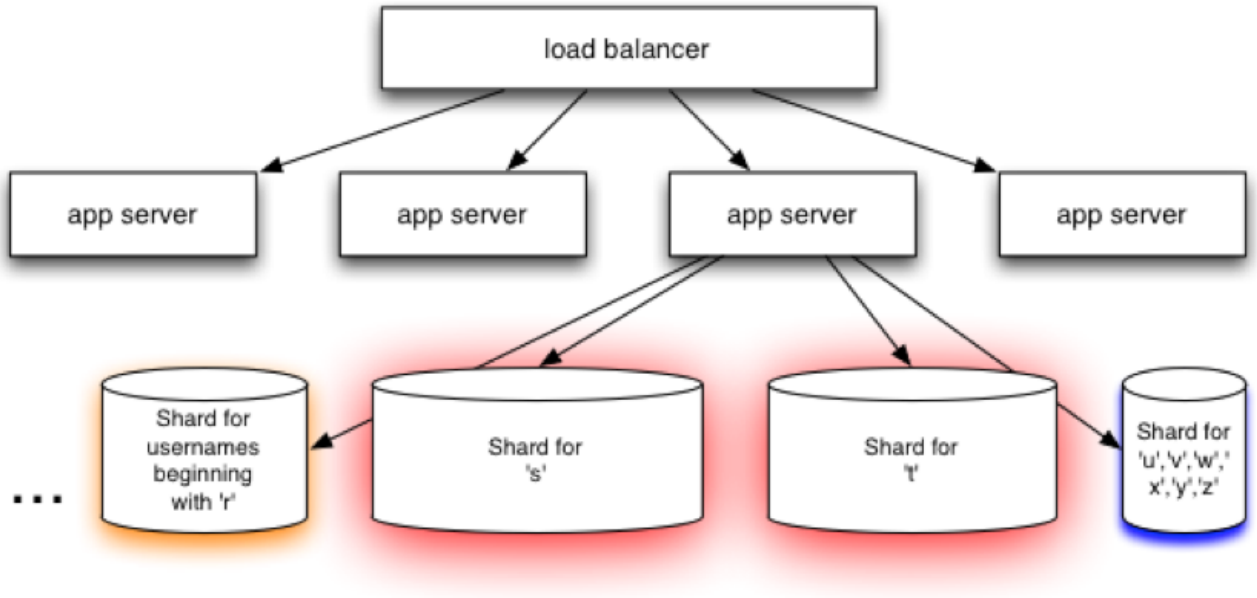
In my last post, I pointed to the LiveJournal model as an example of sharding on which many recent Internet companies have based their own implementations. To understand the design decisions of a sharded environment, let's discuss the differences between two strategies: algorithmic sharding and dynamic sharding.

The Rigidity of Algorithmic Sharding

In algorithmic sharding, each client accessing the shards can determine which shard has the data it needs without contacting another service. For example, algorithmic sharding is used when each shard handles a letter of the alphabet and a user's data is located on the shard that corresponds to the first letter of the username. A common example of algorithmic sharding is a print multi-volume encyclopedia. (I may be dating myself with that reference!)

In this scenario, some shards will have more data on it, and thus more requests to it. This means more scalability problems on popular letters and underutilized hardware on less popular letters. The next hack is often to combine several less-popular letters into the same bucket or split larger letters by the first two or three letters. Some even try reversing the string or adding their ascii values and using a modulo, but they also find similar hot spots. It's much easier to evenly distribute an integer sequence without gaps, but that would require a service shared by all – which is dynamic sharding, not algorithmic.

Fig 1. Algorithmic Sharding



A balanced algorithm can be achieved for any specific dataset at a single point in time, but it is impossible to predict how a user base will evolve. Early social networks Orkut and Friendster became popular in Brazil and the Philippines, respectively, with a large influx of new login usernames based on common names/terms in a language other than the services' original target audience. You will always have to modify the algorithm at some point(s) in the growth of your data. This is where algorithmic sharding gets painful.

Here's a quick story to illustrate just how painful algorithmic sharding can get. When I arrived at CriticalPath in 1998, the company was using algorithmic sharding to partition email boxes across multiple backend servers, and there were obvious hot spots. Changing the algorithm required moving the data of many different users to a new location – a non-atomic operation that takes longer than a normal maintenance downtime and results in periods where only new or old mail was visible to the user (even Consistent Hashes have this problem). Even worse, if any of the mailservers in production had different algorithms, messages would be directed to a volume on which the user did not exist, thus we ran the risk of bouncing messages as undeliverable and potentially unsubscribing email addresses from mailing lists.

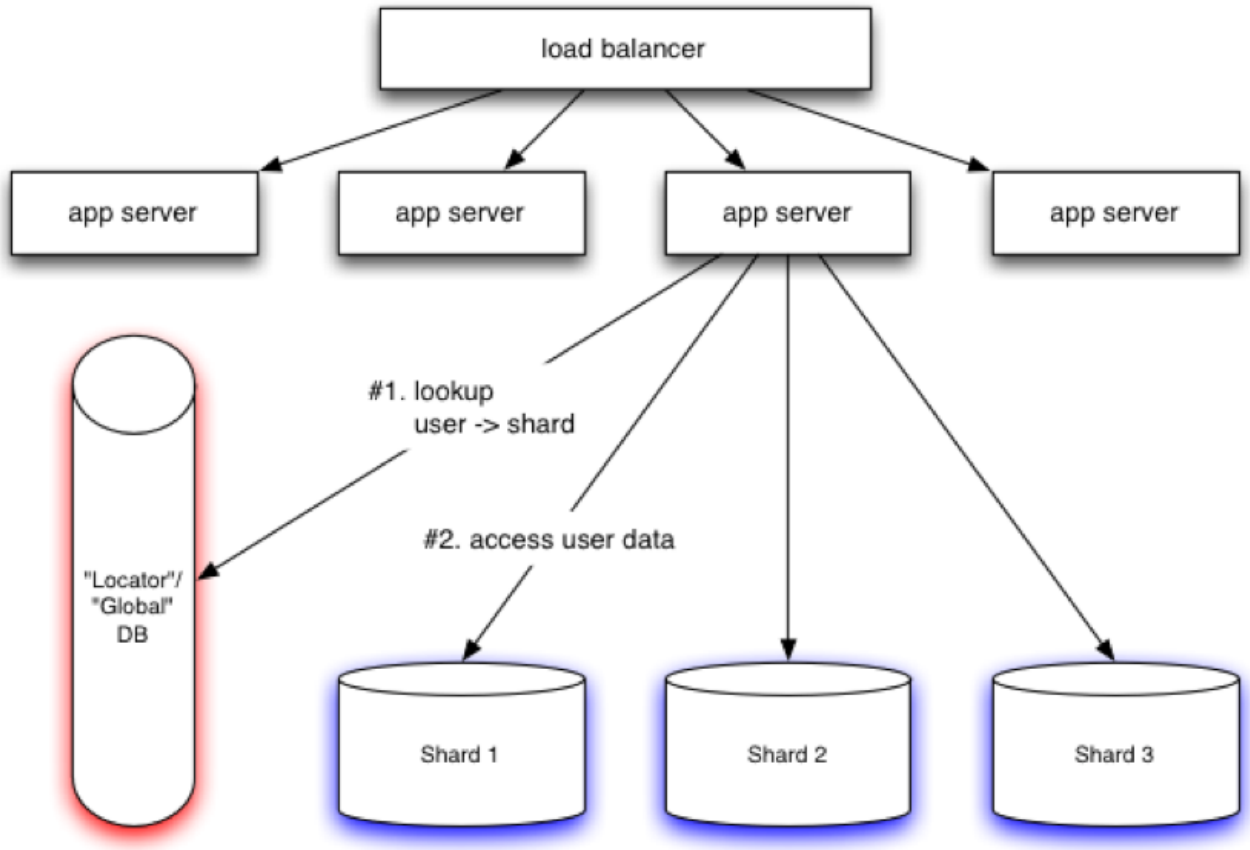
The main problem with algorithmic sharding is that the most common use case ("Does this object exist?") depends on every application server having the same version of the phonebook, so to speak. If this isn't the case, data inconsistencies can result. Many operations teams entrust this homogenous-version problem to configuration management software like cfengine, puppet or chef, but typos in those config files can still cause a development machine to be incorrectly assigned to a production load balancer pool.

Dynamic Sharding Offers Finer Granularity, but re-introduces Dependency

Dynamic sharding addresses the problems of algorithmic sharding by introducing a service that arbitrates existence in a namespace (e.g. usernames), provides the location/shard where the object's data resides, and can even lock an object while it is being moved. This provides the ability to relocate users individually, as opposed to large groups of users, from one shard to another to relieve hot spots.

At CriticalPath we called this service the "Locator" and at SixApart we called it the "Global DB." Expanding on my earlier encyclopedia metaphor, this service is akin to an index volume.

Fig 2. Dynamic Sharding



Keep in mind that this service only contains a few small columns, whereas the bulk of user data is kept in the shards. The fundamental limit on the size of a database is the size of its underlying file(s), and how you approach that limit is simple multiplication: either many thin rows or fewer wide rows. Factors like write load and long recovery and ALTER TABLE times make for a practical size limit much lower than the max file size.

In the case of the Locator or Global DB, the tables should be so simple that they never needed altering, and account creation/deletion/moving is infrequent compared to active users updating their data on the shards.

Contacting this service first can introduce a little latency on some operations, but it's generally considered a good trade and can be mitigated by caching like memcached. But power failure events can happen, resulting in cold caches and scrambling engineers that must throttle the full load that's crippling the single Global DB.

Will Your Shard-Mapping Database Scale?

During my time at SixApart, the TypePad architecture was redesigned to be like LiveJournal, i.e. sharded. The initial sharding process took the better part of an entire year for a team of several engineers who had to determine how each table should be sharded. For example, whether a comment should be stored on the shard where the commenter's data resides or on the shard where the original post's author resides.

The best decision wasn't always clear. So, if the size of the table wasn't causing problems, we kept the entire table on the Global DB, unsharded. New application features often introduced new data that required sharding consideration, and a few months into this process, our Global DB was overloaded and the Misc DB was born – yet another vertical silo of data that was difficult to shard. After all the time and money invested in sharding, it remained a constant exercise that was often so complex that the original vision of the sharding architect would not survive after being passed over to other engineers.

SixApart and CriticalPath were two pioneers of sharding, but many companies are still sharding exactly like this today.

If you are already sharding, consider using Clustrix as a Global DB replacement in your sharded environment. This way, the load on that single point won't force you to constantly re-implement sharding for every new feature object/table.

In my next post in the series, I'll discuss the inefficiencies commonly seen on horizontally scaling shards themselves. Stay tuned!

Part One: A Brief History of Sharding (https://www.clustrix.com/bettersql/sharding-theory-practice-part-one/)

Part Two: The Differences Between Algorithmic and Dynamic Sharding (https://www.clustrix.com/bettersql/sharding-theory-practice-part-two/)

Part Three: What's in a Shard? (https://www.clustrix.com/bettersql/sharding-theory-practice-part-three/)

Part Four: Using Memcached (https://www.clustrix.com/bettersql/sharding-theory-practice-part-four/)

Part Five: The Data Warehouse (https://www.clustrix.com/bettersql/sharding-theory-practice-part-five/)