Sharding: In Theory and Practice (Part Three)

Standard / by Neil Harkins (https://www.clustrix.com/author/neil-harkins/) / January 29, 2013 / No Comments (https://www.clustrix.com/bettersql/sharding-theory-fault-tolerance/#respond)

Part Three: What's in a Shard?

In the first two posts of this series, I offered a perspective on the origins of database sharding and described the architectural problems with algorithmic sharding that led LiveJournal and TypePad to use dynamic sharding to scale. The next challenge of a sharded architecture is adding fault tolerance.

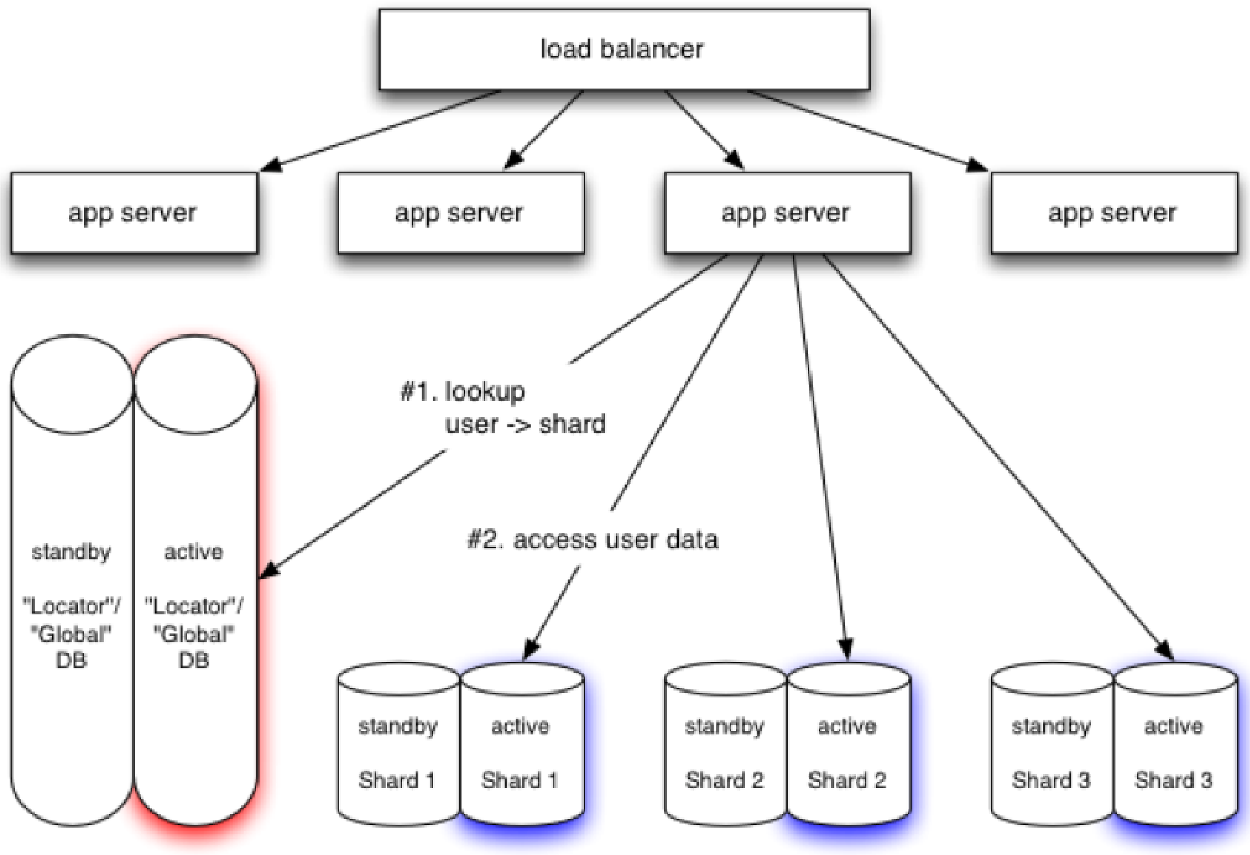
First, let's walk through a typical database failure in order to remind ourselves why we want fault tolerance. After sharding, we now have one global database and multiple shard databases with different failure modes.

If the global database service ever goes down for any reason, objects can't be located and the entire site will go down. In this event, you'd want your load balancer to temporarily redirect all your URLs to a page that explains the outage and gives an estimate of how long it will be so that you don't overload your support department.

If a shard database goes down, then only part of the site will be unavailable. But in order to maintain a similar level of communication regarding the outage with those affected, special handling is necessary in the application code to determine if a user is affected or not.

Ideally, the global database and all the shard databases are always available, so every logical database is commonly a pair of database machines configured in master/master replication. A vip floats between the two sides of each pair, and only one side is active at any one time. This is to guarantee that either side has the capacity to handle the entire load of the other in the event of a failure.

Dynamic Sharding with Active/Passive pairs for Fault-Tolerance



This is somewhat similar to RAID 10 configuration for storage systems; however it does not provide ACID guarantees across a failure since MySQL replication is asynchronous. A failure could occur between the time that an event has synced to disk on one side and the time that its partner gets the event through the binlog and syncs it to its disk.

Alert Fatigue: The Toll of Seconds_Behind_Master

If your application is strictly OLTP and you've aggressively sharded any hot spots, then you may be able to keep the data fairly synchronized between the two sides of the pair. However, the larger the gap, the greater chance of lost data or conflicts from users who re-submit writes stuck in limbo on the failed side.

MySQL replication takes parallel writes and serializes them, so several things can contribute to slave lag: OLAP-style long-running writes (e.g. pre-computing top commented posts and tag clouds), schema changes that lock the entire table (even if executed outside the replication stream), and more.

Most teams that manage database operations set an alert threshold on the slave's Seconds_Behind_Master value in order to catch the potential for data in limbo before a failure that would make it reality. But when you factor that alert frequency with the sheer number of shards (2x for both replication directions), it's easy to see how an operations team can become exhausted by managing problems rather than building a strong infrastructure ready for the next growth spurt.

Adding Fault Tolerance Shouldn't Double the Price of your Database

Database hardware is one of the most expensive SKUs in your datacenter, and using only 50% of that hardware seems like a waste. Most databases don't provide visibility into how close to the cliff you're standing. In fact, in most cases, adding an additional 100% of your current workload would peg your CPU or IO such that the number of concurrent queries stacks up to the point where the database simply cannot keep up with the query submission rate.

Slaves and Chains: As Oppressive As They Sound

I've seen the architecture diagrams of some well known companies that use replication topologies other than master/master within their shards, but these designs introduce problems in addition to the ones described above.

Read slaves: In an attempt to offload the read load on their shards, some companies use two or more read slaves off one single shard DB. This is ironic since they also use memcached, except they use it for caching HTML instead of caching database rows. If the single shard master goes down, the data is offline until it can be recovered. Re-homing other slaves to a newly promoted slave is not a simple operation, since each slave has its own binlog positions based on the time the slave was created. These companies have effectively given up fault tolerance for yet another tier to address scalability.

Uni-directional circular chains: This design attempts to use all of its hardware, but also increases the latency between a write of some data happening on one node and that same data being readable on the node furthest away on the chain. In the event of a single break in the chain, the chain becomes linear rather than circular, and all writes must then go to the top of the chain. Otherwise, some of the instances in the chain simply won't get the writes until the full loop is repaired.

Building a Better Shard

Clustrix has built-in fault tolerance, including synchronous ACID guarantees across failures, and supports online schema changes – making us an excellent replacement for a shard. Our base system consists of three nodes, where the failure of any single node triggers the other two nodes to handle an additional 1/6th of the overall load. As you increase the number of nodes in the cluster, the impact of a failure decreases to a negligible amount. In other words, the failure of one node in a ten-node cluster results in a mere 1.1% increase. So, reclaim your idle hardware! Eliminate replication from within your shards and all the unnecessary busy work it creates.

In the remaining posts in this series, I'm going to discuss database caching layers and reference a data warehouse where the shards can be recombined. 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/)