Sharding: In Theory and Practice (Part Four)

Standard / by Neil Harkins (https://www.clustrix.com/author/neil-harkins/) / February 11, 2013 / No Comments (https://www.clustrix.com/bettersql/sharding-theory-using-memcached/#respond)

Part Four: Using Memcached

Welcome back to our blog series on database sharding. As I mentioned in **part one (https://www.clustrix.com/bettersql/sharding-theory-practice-part-one/)** of this series, memcached was invented at LiveJournal, and its purpose is to reduce the number of redundant reads hitting their databases. LiveJournal observed that 80% of the traffic accesses only 20% of the data, with the rest being a “long tail” of infrequently accessed data, as is common for most Internet services. Blogs are continuous streams of data, constantly growing, but that popular 20% is a reasonable amount to keep in Random Access Memory, where it can be served much faster than slower, nonvolatile storage.

Modern operating systems cache reads from filesystem devices in otherwise unused memory, but processes on the system get priority when they need memory. Databases in particular need a good amount of temporary memory for assembling query results and sending them to clients. Because of this, entries from the FS-Cache often get evicted and reads for the data must go to the much slower storage device. There is currently no way to partition memory such that a fixed amount is committed to the FS-Cache. And even if there were a way, there would be evictions as long as machines have more storage than memory.

As I explained in part three, **“What’s in a Shard?” (https://www.clustrix.com/bettersql/sharding-theory-practice-part-three/)** the limitations in MySQL replication often force a decision between fault tolerance and scalability of reads (MySQL offers no solution for scalability of writes, hence sharding). Unlike some companies, LiveJournal was wisely unwilling to give up fault tolerance, so they built a separate caching layer.

In-Band vs. Out-of-Band Caches

An in-band cache is a drop-in replacement for a none-the-wiser application, where the caching layer acts as a middleman and looks at the request to see what's being written (update the backend and then the cache), or read (query the cache first, and if not there, query the backend). The Linux FS-Cache is an example of this, but the block device API is much simpler than a SQL parser (trust us, we've written one).

This is why LiveJournal built memcached as an out-of-band cache. The application talks to a memcached daemon and then talks to the database – both in separate connections.

This simple daemon was written in a way that allocates a chunk of memory as a process, listens on a TCP port for simple GET and SET commands to a simple flat hash table in that chunk of memory. It will evict entries when it gets too full, and also provide stats on that frequency so operators know when to add more instances of memcached.

How Does an Application Know Which Memcached Instance to Use?

This question sounds eerily similar to the overall problem of this entire series – how to scale from one single server to multiple servers. As you see, the same problems are seen repeatedly in sharding, and the same tools are commonly used.

Livejournal chose algorithmic sharding for memcached. If you recall from my second post in this series, algorithmic sharding was not a good choice for the sharding layer. However, because caches always have the fallback to the actual datastore, the missing data problem simply does not exist. However, the inverse problem of stale data can still happen if some of the application servers are not using the same algorithm.

How Memcached is Used

To use this new caching daemon, LiveJournal needed to implement application changes. Prior to memcached, the application might have contacted the database like this:

```
SELECT * FROM users WHERE pk = 123456;
```

But that query could be replaced by a memcached request like this:

```
get TABLE/user/PK/123456
```

If the key exists in the memcached instance, then the data portion will contain the column values (serialized so individual columns can be extracted). If the key doesn't exist in memcached, the application performs the point select against the database and will then populate memcached with the result.

LiveJournal addressed range selects and joins by changing the queries to return only primary key values, thereby avoiding the hidden joins from an index to the full table, and essentially asking the database to send less data back across the network. But again, if the row did not exist in memcached, then it would be requested from the database. And with multiple rows, we multiply the number of round trips. So, when the data is not already in memcached, the load on the database is much greater than it would be without using memcached at all!

The Pain of Cold Caches

To mitigate this problem, memcached changes generally roll in during off-peak hours in order to warm up the cold, empty caches. And normally, they stay nice and toasty warm 24/7, but an occasional data center power outage will wreak havoc on your memcached cluster.

Unfortunately, one such outage hit SixApart’s properties. When power was finally restored, memcached instances were completely cold. The normal amount of requests to the database was multiplied to database-crushing proportions, prolonging our downtime. We were effectively performing a denial of service attack on our own servers. I was forced to use the same defensive techniques we use during a DOS attack, basically throttling web requests in our load balancer until our memcached instances warmed up enough to reduce the database load to normal.

Doesn’t Your Application Have More Important Things to Do?

Because Clustrix is a scalable database, each node contributes memory to the one logical database with one logical in-band cache, achieving far better cache utilization than a conventional single instance database. You also gain more storage and computational power – something memcached nodes don’t necessarily give you.

If your application is already using memcached, you will not experience any compatibility problems with Clustrix, since memcached is out-of-band. Memcached never talks to the database or vice-versa – rather, the application keeps them in sync. However, that extra work requires more application servers in addition to the memcached servers, which increases the datacenter power and cooling costs behind your operating expenses.

In the next post, I’ll cover data warehouse techniques for sharding. 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/)