# Troubles with Sharding - What can we learn from the Foursquare Incident?

Friday, October 15, 2010 at 8:15AM

Todd Hoff in Strategy, postmortem, sharding

For everything given something seems to be taken. Caching is a great scalability solution, but caching also comes with problems. Sharding is a great scalability solution, but as Foursquare recently revealed in a post-mortem about their 17 hours of downtime, sharding also has problems. MongoDB, the database Foursquare uses, also contributed their post-mortem of what went wrong too.

Now that everyone has shared and resharded, what can we learn to help us skip these mistakes and quickly move on to a different set of mistakes?

First, like for Facebook, huge props to Foursquare and MongoDB for being upfront and honest about their problems. This helps everyone get better and is a sign we work in a pretty cool industry.

Second, overall, the fault didn't flow from evil hearts or gross negligence. As usual the cause was more mundane: a key system, that could be a little more robust, combined with a very popular application built by a small group of people, under immense pressure, trying to get a lot of work done, growing really fast (3 million users, 200 million checkins, 18,000 new per day), deprioritized a few steps that eventually turned into a cascading series of failures.

Was it preventable? Yes, but it was also understandable. Current systems from the dominant paradigm are stable precisely because they have been through years and years of these kind of war stories and have been hardened over time into an enviable robustness. Whatever weaknesses they do have, everyone just "knows" how to route around using tools and tricks that have by now become second nature. NoSQL hasn't gone through this maturation process yet. Tools, tricks, and best practices are still being developed. NoSQL and other new approaches will have to follow the same lifecyle, but hopefully by this kind of public disclosure and discussion we can really jump down that learning curve.

## What Happened?

The problem went something like:

- Foursquare uses MongoDB to store user data on two EC2 nodes, each of which has 66GB of RAM (high-memory quadruple extra large instance). Data replicates to slaves for redundancy. MongoDB is a document oriented database which stores data in JSON format, indexes can be made on any field, and it supports auto-sharding.
- Users, written to the database, did not distribute evenly over the two shards. One shard grew to 67GB, larger than RAM, and the other to 50GB.
- Performance tanked once physical RAM was exceeded. Memory started being paged to disk as a way to free up memory for hotter data to be brought into memory. But disks are slow for random access, and the whole system slowed down to disk speed. Queries became slow, which caused a backlog of operations as more and more queries stacked up, which caused even more paging to occur, which brought the site down.

- A third shard was added, yet the first shard was still hitting disk because not as much memory was freed as expected. The reason was Foursquare checkins are small, 300 bytes, MongoDB uses 4K pages, so when you move a checkin the 4K page will still be allocated, the result is no memory actually freed. Memory will only really be freed when all data is moved off a page.
- MongoDB has a compaction feature, which was turned on, but compaction was slow because of the size of the data and because EC2's network disk storage, EBS, is relatively slow.
- The solution was to reload the system from backups so the data was resharded across the three shards. All data now fits in physical RAM.

The number of issues this scenario brings up is astonishing. They range from low level cache and system design, questioning the ability of cloud IO to support high performance applications, deciding on the role key infrastructure software should play in it's own management, arguing over what standards of professionalism should be applied to quick moving startups, proposing how severe system problem should be handled in this litigious age, to if handling a problem well should outweigh in a user's mind the problem in the first place. A lot to consider, but we'll stick to the more technical aspects of sharding.

## What is a Shard?

If we are going to get a better feel for how shards work and how they can fail, we first need to know what they are. I'm going to take the traditional definition of shard and generalize it a bit:

1. **shard**: a subset of a data set that is stored and accessed on a particular node.
2. **sharding**: the process of creating and managing shards to properly

match data size to resource usage and availability. Resources can include: CPU, memory, disk, network, performance and reliability.

Some miscellaneous notes and elaborations: With sharding an entire data set is split into chunks and spread horizontally across a set of nodes, this is horizontal partitioning. The data set can be fixed in size or grow continually. Data and indexes can be completely in-memory, primarily on disk, indexes primarily in-memory and data on disk, or it can be a goal to have a working set can be in memory and the rest on disk. If your algorithms walk across a lot of pages or use a lot of historical data, then a RAM only and a working set approach may not work well. Databases can use the operating system to manage paging to disk when the data is larger than available RAM or the database can manage on disk storage itself. The format of the data and the operations allowed on the data are system specific. Systems vary a lot in the data types they support:  BLOBs, documents, JSON, columns, and references as first class citizens. Systems vary a lot in the operations they support: key-value lookups, increment operators, map-reduce operations, queries, limited transactions, set operations, searching, secondary indexes, auto-sharding, monitoring, REST API, and a management UI. The sharding process can be completely manual or be under different degrees of automation. Some systems take a few configuration parameters at startup and the system is under manual control after that. Some systems offer a high degree of automation that removes much of the responsibility off of developers. Usually it's somewhere in between.

Data is typically sharded when it no longer "fits" on one node. We generally think of sharding as a means to get around RAM or disk limitations, but it's a more general principle than that, it's a way to get around all types of limitations by matching the data size to a node such that there are enough resources to do whatever needs to be done with the data. Limitations can be any combination of:  CPU, memory, disk, network, performance and reliability.

Some examples. The common case is for a shard to grow and use too much RAM, so the shard must be managed (split, moved, etc) to remove the RAM constraint. It's also possible for all your data to fit in RAM, but to not have enough available CPU to operate on the data. In that case a shard would have to moved, split, etc so that the CPU constraint is removed. It's possible to have enough RAM, but not enough reliable network to perform the required number of IO operations. In this case the data must be moved, split, or replicated such that the IOPS target can be reached.  And so on. They key is to move the data around in such away that constraints are removed and performance targets are met.

**The goal of sharding is to continually structure a system to ensure data is chunked in small enough units and spread across enough nodes that data operations are not limited by resource constraints.**

A good product match for you is one that accomplishes this goal in a way that matches your expectations, capabilities, and budget.

## A Trouble Typology for Shards

Sharding is often seen as the secret sauce of scaling, horizontally inclined nirvana, but there are possible problems in paradise not everyone may have fully considered. We need to build up the same sort of "knowing" about potential problems with sharding that we have built up for RDBMSs. Here's a first cut at a typology of sharding problems to look out for. It's in no way meant to be an exhaustive list or a great explanation of all the issues. It's just a start.

- **Node Exhaustion**. The capacity of a node is exceeded. The shards on a node have caused resource exhaustion in one or more dimensions: memory, disk, network, CPU, etc. An example is when an in-memory based system runs out of RAM to store data and/or indexes.

- **Shard Exhaustion**. The capacity of a shard is exceeded. Shards are often fixed sized. If data is continually written to a fixed sized shard, or a particular value becomes too large, the shard will exceed its capacity. Imagine a User object that stores the social network for a popular user, it could have millions of entries, which could easily blow out the resources for a shard.
- **Shard Allocation Imbalance**. Data is not spread evenly across a shard set. This happens when keys hash into one shard more than others. Key examples: date, userd ID, geo-location, phone number. It's difficult for developers to know the distribution of a key apriori, so a disproportionate amount of data can be written to just a few shards while the other shards are left relatively empty. This can lead to shard and node exhaustion.
- **Hot Key**. Excessive access for a particular key or set of keys can cause node exhaustion. Imagine data for Justin Bieber, Lady Gaga and Ken Thompson all ending up on the same shard, that node would become overwhelmed with read and write accesses.  Even in-memory systems can suffer from Hot Key problems, Facebook has seen this in their memcached clusters.
- **Small Shard Set**. Data is stored on too few shards. When only a few shards are used all the data ends up on just one or two shards, which can easily lead to hot key and exhaustion problems. It also makes live repair harder because the systems being repaired are the same ones that are failing. Management operations, to say split a shard, may not be received by a failing node because the node is too busy. Even if the management operations are accepted the node may not have enough CPU or RAM to carry out the operations. Using more shards reduce the amount of data on each shard which makes for quicker and more predictable operations on shards.

- **Slow recovery operations**. Correcting problems is slow. Many problems can be masked by a speedy recovery, which can keep MTTR low enough to be acceptable. Many systems will be slow in recovery operations which reduces the reliability of your system. For example, is your IO system fast enough to reload all your data from backup within your SLA?
- **Cloudbursting**. Resources can't be spun up fast enough to respond to load spikes. Load can be very spiky in nature, increasing greatly in just a few seconds, yet it can take many minutes to spin up instances to handle the load. How quickly can your system respond to cloudbursts?
- **Fragmentation**. Effective available memory becomes significantly less that total available memory. It surprisingly tricky to manage memory of all different chunk sizes while it is being rapidly allocated and deleted. Memory gets fragmented. A 100 byte allocation that was actually allocated from a 100 byte pool when a system started could over time be allocated from 2K pool later and all that memory goes to waste. Then a garbage collection pass is necessary to rationalize the pool again and while garbage collection is happening, the system generally can't serve queries.
- **Naive disaster handling**. When a problem occurs the system dies instead of handling it gracefully. Operations is another area where products may be less advanced than expected. For example, when you add shards are you required to add all the data again?
- **Excessive paging**. Letting the OS manage virtual memory through paging can lead to page thrashing and when a system starts thrashing it's pretty much over. Very little meaningful work is being done because the system is spending all its time paging. A paging system pages "cold" blocks out of memory to disk and brings in hot blocks off disk into memory. How does an OS know what's hot or not? It really doesn't, and that's why a lot of products choose to explicitly manage their storage system by storing records on disk

in some sort of database when the records aren't in memory.

- **Single Points of Failure**. A feature of a design where if one part of the system fails then the whole system fails. This often occurs in cluster coordination and management, where one node is special and if it fails the cluster fails.
- **Data Opacity**. Data is not observable. Often systems need to respond to data changes. When an attribute changes or thresholds are exceeded, an email may need to be sent or an index may need to be rebuilt. If your data manager doesn't support event listeners (or some other mechanism) then there's no way to build higher level interactions on top of changing data.
- **Reliance on developer omniscience**. The database requires the developer to make a lot of decisions upfront that are hard to make. Partitioning algorithms is one example.  Which key should you hash on? How many shards should you use? How big should each shard be? How many nodes should be used? Should you use random partitioning or sequential partitioning, or something else? The problem is developers really don't have good answers to these questions. How can a developer capacity plan when they know so little about how their application will be used? How can a developer determine their working set size when it's very application dependent? What if you are wrong and not even all your indexes can fit in RAM? It means there will be a lot of disk contention from multiple concurrent clients, which causes everything to backup and start performing. Does your system help you with these decisions? Does it make it easy to fix a mistake? Can it help you know when an assumption that was made has or will be violated?
- **Excessive locking**.  Locks are taken to prevent simultaneous access to shared data. The more locks generally the slower a system will be as threads are blocked waiting for the locks to clear. As data structures get larger locks are taken longer as it takes longer to

iterate and operate on lists. Often yields are inserted to give other threads a chance to run, but yields are sign that a design needs to change. Garbage collection can be another source of excessive locking that blocks queries from being executed. Does your system take long locks, especially as more data is stored or concurrency goes up?

- **Slow IO**. When the data does not fit in RAM, then the speed of the IO path is crucial for normal operations and for recovery operations. Foursquare uses EBS, which is network attached storage, which makes the IO path travel through CPU, the network, and disk. It may seem like Foursquare use using only a gig more than their available RAM, but that would only be true if the memory was been managed that way. When virtual memory and paging are involved that memory can be paging in and out of anywhere. Foursquare found that *EBS read rates were not sufficient if the data + indexes didn't fit into RAM*.
- **Transaction model mismatch.** Figure out where your database falls on the CAP question and see if that matches your needs. Do you require transactions or is eventual consistency OK? Can the database run in different availability zones and does that matter to you?
- **Poor operations model**. Figure out how easy it is to test, deploy, manage, and optimize your database. Is there a UI? Is it command line based? Does it do all the things you need to do? Is it finicky or does it just work? Are you doing all the work or does it do the work for you?
- **Insufficient pool capacity**. Elasticity requires resources be available so they can be allocated on demand. In some availability zones, especially at certain times of the day, it can be difficult to find nodes of a desired capacity.
- **Slow bulk import**. Does your system support fast importing of bulk data? If not, it could take you hours to upload large amounts of data

from another system or restore from a backup. Many systems suffer from this problem and it can be quite surprising.

- **Object size and other limits**. Most systems will have limits on the size of the objects they can store in order to ensure bounded latencies on operations. Make sure that these limits match your needs or that you design your schema accordingly.
- **High latency and poor connection sensitivity**. If you plan on operating across datacenters or on poor networks, verify that your system can handle that scenario. Operating under high latency conditions or lossy network conditions requires special design that if not present will cause a lot of pain.

I'm sure there are a lot more potential troubles out there. What do you think?

And what can you do with all this information? Check how your system handles these issues and if you are happy with how they are handled.  If not, what can you about it?

## Possible Responses to Troubled Shards

What can be done about the troubles? Some possible options are:

1. **Use more powerful nodes**. Obvious, but scaling-up is often the best solution.
2. **Use a larger number of nodes**. Using fewer nodes makes you much more susceptible to problems on those nodes.
3. **Move a shard to a different node**. If that node has some headroom then your performance should improve. How easy is it to move shard around with your system? Is the bucket model virtualized so it's relatively easy to move shards around?

4. **Move keys from a shard to another shard or into its own shard**. A very hot key needs to placed into a shard where it can get the resources. Can your system move individual keys around? What happens if an individual key requires sharding itself?
5. **Add more shards and move existing data to those shards**. Can the data be moved fast enough to fix the overload problems? Are requests queuing up? Will enough memory be freed on the overloaded system or will fragmentation issues prevent the memory from being reclaimed?
6. **Condition traffic.** This is just smarter handling when problems occur. Prioritize requests so management traffic can be received by nodes even when the system is thrashing. Requests can be dropped, prioritized, load balanced, etc rather than being allowed to take down an entire system.
7. **Dark-mode switches**. Have the ability to turn off parts of system so load can be reduced enough that the system can recover.
8. **Read-only disaster mode**. Have a read-only option so the system can fail to a mode where your system is accessible for reads, even if it can't handle writes, while repairs are going on.
9. **Reload from a backup**.
10. **Key choice**. Use a uniform distribution mechanism to better spread keys across shards. Select a better key and reshard.
11. **Key mapping**. Mapping data to a shard via hashing is quick and efficient, but it's not the only way to map keys to shards. Flickr, for example, uses and index server that directly maps individual users to a shards. Should developers ever even need to now about shards?
12. **Replicate the data and load balance requests across replicas**. Does your database support selectable consistency policies? Can some of your data use a more lenient policy? Does your code support using read replicas when the main servers are down?
13. **Monitor**.  Monitor resource usage so you can get a heads up and take preventative action. This is obvious and was hammered on

pretty hard by commenters. Foursquare knew they should monitor, but didn't have the time. A good question is if this is something they need to build from scratch? Isn't there a service the connects MongoDB and EC2 together without a lot of fuss and bother? MongoDB does provide such statistics. Look for a spike in disk operations per second, which would indicate increased disk activity. Look for increased request queue sizes and request latency, both of which indicate something is going wrong in the core processing engine.

14. **Automation**. Use a system that has more automated elasticity, sharding, and failure handling. Can it connect into systems like RightScale or Amazon's AutoScale?
15. **Background reindexing and defragmentation**. Give consideration to query latency during these operations.
16. Capacity plan. Better estimate your system requirements. Continually capacity plan to continually adjust resources to fit projected needs. The growth rate for Foursquare was predictable, so it shouldn't have been a complete surprise that at some point in the growth curve more resources would be needed. This is where capacity planning comes in. Difficult for a startup, but it would nice if this functionality was part of a general dashboard system.
17. **Selection**. Select a system with memory management policies and other features that fit your use case. If you know you won't have time to manage a system then select a system that will do more of that for you. If you aren't sure about key distribution pick a system that helps you with that. If you aren't sure about your working set size then pick a system that manages that in a robust fashion.
18. **Test**. Test your system under realistic conditions. Test failure and recovery scenarios. Test rapid allocation/delete scenarios. Test mixes of allocation sizes. Test query response times during failure scenarios, garbage collection, and management operations like adding shards. Does everything still work as promised? Integrate

testing into your build system so it's possible to know when any key metric changes. Again, this advice is from Captain Obvious, but in the cloud this is all doable. If you don't test your customers will be testing for you.

19. **Figure a way to get out of technical debt**. Startups are short on time and people. They are busy. Shortcuts are taken and technical debt is incurred. How can you get out of debt so that features aren't dropped on the floor? Look for services that can be integrated via a simple API. Hire contractors. Don't be so picky about who you hire. Think about it at least. Maybe there's a creative solution other than pushing it out to a much later release.
20. **Better algorithm selection**. Use, when possible, incremental algorithms that just need an event and a little state to calculate the next state. A common example is when calculating an average use a running average algorithm rather than one that requires all values for for every calculation. One of the problems Foursquare had was they use a person's entire check-in history to award badges. This means they have to store a lot of "useless" data that will definitely kick in paging while  walking these data structures. It may be better to drop a feature if it can't be calculated efficiently.
21. **Separate historical and real-time data**. Historical data could be kept in a different datastore so that it didn't interfere with real-time operations. Then background calculations can be run that update the real-time system without impacting the real-time system.
22. **Use more compact data structures**. Compact data structures will use less memory which means more data can fit in memory.
23. **Faster IO**. If you are expecting that all your data will not fit in RAM, then make sure your IO system isn't the bottleneck.
24. **Think about SPOF**. Single points of failure are not the end of the word if you know about them and work it into your MTBF calculations. The point is to know they could be a problem and determine how the database you are using deals with them.

25. **Figure out how you will handle downtime**. Let people know what's happening and they will still love you. Foursquare now: tweets status messages, provides regular updates during outages, added a status blog, and created a more useful error page that had more information on it. All good.

Again, I'm sure there are lots more possible solutions. What are yours?

Clearly no system is perfect or will be used perfectly. The point here is really just to consider what issues are out there, bring up some points you may need to think about, and think about how you might fix them.

## Related Articles

- Guidelines for Handling Ports-mortems from Transparent Uptime.
- Good Hacker News Thread and a great Hacker News Thread
- Related Thread on MongoDB Email List and another Related Thread
- Foursquare MongoDB Outage Post Mortem
- The Google App Engine folks are also great with their post-mortems.
- What's wrong with 2006 programming? by Antirez of Redis. A great explanation of why using an OS based paging system for cache management is not always a good idea. A great discussion in the comments.
- A Cache-Oblivious Implicit Dictionary with the Working Set Property by Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen.
- Learning from Others by Dan Pritchett
- CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data by Sage A. Weil, Scott A. Brandt Ethan, L. Miller

Carlos Maltzahn. *CRUSH is designed to facilitate the addition and removal of storage while minimizing unnecessary data movement.*
- Foursquare and MongoDB: What If by Jeremiah Peschka

---