**Sharding Pitfalls**

**1. Using a monotonically increasing shard key (like ObjectID)**

Why is it a bad idea?

The short answer is insert scalability. If you select such a shard key, all inserts (new documents) will go to a single chunk - the highest range chunk, and that will never change. Hence, regardless of how many shards you add, your maximum write capacity will never increase - you will only ever write new documents to a single chunk and that chunk will only ever live on a single shard.

Occasionally, this type of shard key can be the correct choice, but if so then you won’t be able to scale for write capacity.

**2. Trying to Change Value of the Shard Key**

Shard keys are immutable (cannot be changed) for an existing document. This issue usually only crops up when sharding a previously unsharded collection. Prior to sharding, certain updates will be possible that are no longer possible after the collection has been sharded.

Attempting to update the shard key for an existing document will fail with the following error:

cannot modify shard key's value fieldid for collection: foo.foo

**3. Waiting too long to add a new shard (overloaded)**

You sharded your database and scaled horizontally for a reason, perhaps it was to add more memory or disk capacity. Whatever the reason, if your application usage grows over time so (generally) does your database utilization. Eventually, your current sharded cluster will pass a certain point, let’s call it 80% utilized (as a nice round estimate), such that it becomes problematic to add another shard. Why? Well, adding a new shard to a cluster is not free, and it is not instantaneous. It consumes resources and (initially) accepts very little traffic.

Essentially, at the start of its existence, a newly added shard costs you capacity instead of adding capacity. The length of time it will stay in this state will depend on the balancer and how long it takes for a significant portion of “busy/active” chunks to move onto the new shard.

It can often be easier to visualize this process, so let’s make up some hypothetical numbers and set the bar relatively low. Our imaginary existing cluster will be a set of 2 shards, with 2000 chunks (500 considered “active”) and to that we need to add a 3rd shard. This 3rd shard will eventually store one third of the active chunks (and total chunks). The question is, when does this shard stop adding overhead overall and instead become an asset?

In reality, this will vary from cluster to cluster and have a lot of dependencies and variables - in other words you need to have good metrics about your cluster, particularly your load bottleneck.

Therefore we will once again use our imaginations and go with a relatively low bar: when 5% of active chunks–that is, those chunks seeing most traffic–have migrated to the new shard, you should expect a net gain in performance. In our imaginary system we have evaluated our load levels, the expected impact of migrations and have determine that once that 5% threshold of active chunks has been migrated to the new shard it can be considered a net gain for the overall system. Once all chunks have been balanced, then the migration overhead disappears, but initially this will be an expected trade off.

**4. Under-provisioning Config Servers**

Provisioning enough resources without being wasteful is always tricky, and all the more so in a complicated distributed system like a MongoDB sharded cluster. Everyone wants to use their hardware, virtual instances, virtual machines, containers and the like in the most efficient way possible, and get the best bang for their buck. Hence it is only natural to take a look at the various pieces of a distributed cluster and look for lower utilized pieces that could be put on less expensive resources.

The most common pitfall here with MongoDB are the config servers, which are often neglected when stress testing a cluster. In testing environments and smaller deployments (unless specific measures are taken to stress them) they are relatively lightly loaded and usually identified as candidates for lesser instances/hardware.

The problem is that these are critical pieces of infrastructure. They may not be heavily loaded all the time, but when they do see load and struggle to service requests, that can impact all queries (reads, writes, authentication) and add latency to all requests made of the cluster in question.

In particular, the first config server in the list supplied to your mongos processes is vital. This is the config server that all mongos processes will default to read from when fetching or refreshing their view of the data distribution in your cluster. Similarly, this is the server that will be hit when attempting to authenticate a user. If it is under-provisioned and cannot service queries, or if it has problems with networking (packet loss, congestion), then the effects will be significant.

**5. Using the count() command on sharded collections**

This pitfall is very common, and it seems to hit somewhat randomly in terms of how long someone has been running a sharded environment. At some point, a question will arise along the lines of:

“How are we tracking/verifying/checking how many documents we have in each collection on each shard, how balanced are they and do they agree with <some other system that holds the same data>?”

Hopefully no one is actually constructing questions this way in your organization, but you get the basic idea. The most obvious way to do a quick check on this type of thing is to count the documents and see if the numbers make sense and/or agree with counts elsewhere. That thinking naturally leads people to the [count command](http://docs.mongodb.org/manual/reference/command/count/) and they proceed to use it to gather figures for their documents and collections.

**6. Chunk balancing != data balancing != traffic balancing**

he balancer in a sharded cluster cares about just one thing:

Are chunks for a given collection evenly balanced across all shards?

If they are not, then it will take steps to rectify that imbalance. This all sounds perfectly logical, and even with extra complexity like tagging involved the logic is pretty straight forward. If we assume that all chunks are equal, then we can rest assured that our data is being evenly balanced across all the shards in our cluster and rest easy at night.

Although that is sometimes, perhaps even frequently, the case it is not always true - chunks are not always equal. There can be massive “jumbo” chunks that exceed the maximum chunk size (64MiB), completely empty chunks and everything in between.

Let’s use an example from our first [pitfall](http://blog.mongodb.org/post/98888988013/sharding-pitfalls-part-i), the monotonically increasing shard key. For our example, we have picked just such a key to shard on (date), and up until this point we have had just one shard and had not sharded the collection. We are about to add a second shard to our cluster and so we enable sharding on the collection and do the necessary admin work to add the new shard into the cluster.

**7. Waiting too long to shard a collection (collection too large)**

This is not very common, but when it falls on your shoulders, it can be quite challenging to solve. There is a maximum data size for a collection when when it is initially split which is a function of the chunk size and data size [as noted on the limits page](http://docs.mongodb.org/manual/reference/limits/#Sharding-Existing-Collection-Data-Size).

If your collection contains less than 256GiB of data, then there will be no issue. If the collection size exceeds 256GiB but is less than 400GiB, then MongoDB may be able to do an initial split without any special measures being taken.

Otherwise, with larger initial data sizes and the default settings, the initial split will fail. It is worth noting that once split the collection may grow as needed and without any real limitations as long as you can continue to add shards as data size grows.