

MongoDB Sharding

How queries work in sharded environments

One small server. We want more capacity. What to do?



Traditionally, we would scale
vertically with a bigger box.



With sharding we instead scale horizontally to achieve the same computational/storage/memory footprint from smaller servers.



We will show the vertically scale db and the horizontally scaled db side-by-side for comparison.



A sharded MongoDB collection has a shard key. The collection is partitioned in an order-preserving manner using this key. In this example **a** is our shard key:

`{ a : ..., b : ..., c : ... }` *a is declared shard key for the collection*



$\{ \underline{a} : \dots, b : \dots, c : \dots \}$ *a is shard key*

Metadata is maintained on chunks which are represented by shard key ranges. Each chunk is assigned to a particular shard.


Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

When a chunk becomes too large, MongoDB automatically splits it, and the balancer will later migrate chunks as necessary.



$\{ \underline{a} : \dots, b : \dots, c : \dots \}$ *a is shard key*

`find({ a : { $gt : 333, $lt : 400 } })`



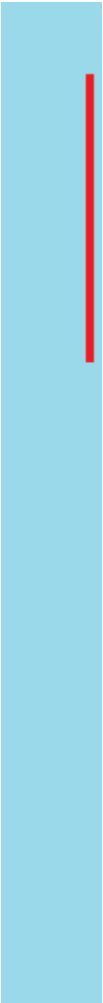
Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

The mongos process routes a query to the correct shard(s). For the query above, all data possibly relevant is on shard 2, so the query is sent to that node only, and processed there.



$\{ \underline{a} : \dots, b : \dots, c : \dots \}$ *a is declared shard key*

find({ a : { \$gt : 333, \$lt : 2012 } })




Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

Sometimes a query range might span more than one shard, but very few in total. This is reasonably efficient.




$\{ \underline{a} : \dots, b : \dots, c : \dots \}$ *non-shard key query, no index*

find({ b : 99 })



Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

Queries not involving the shard key will be sent to all shards as a “scatter/gather” operation. This is sometimes ok. Here on both our traditional machine and the shards, we do a table scan -- equally expensive (roughly) on both.



`{ a : ..., b : ..., c : ... }` *Scatter / gather with secondary index*

`ensureIndex({b:1})`

`find({ b : 99 })`

Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

Once again a query with a shard key results in a scatter/gather operation. However at each shard, we can use the `{b:1}` index to make the operation efficient for that shard. We have a little extra overhead over the vertical configuration for the communications effort from mongos to each shard -- not too bad if number of shards is small (10) but quite substantial for say, a 1000 shard system.



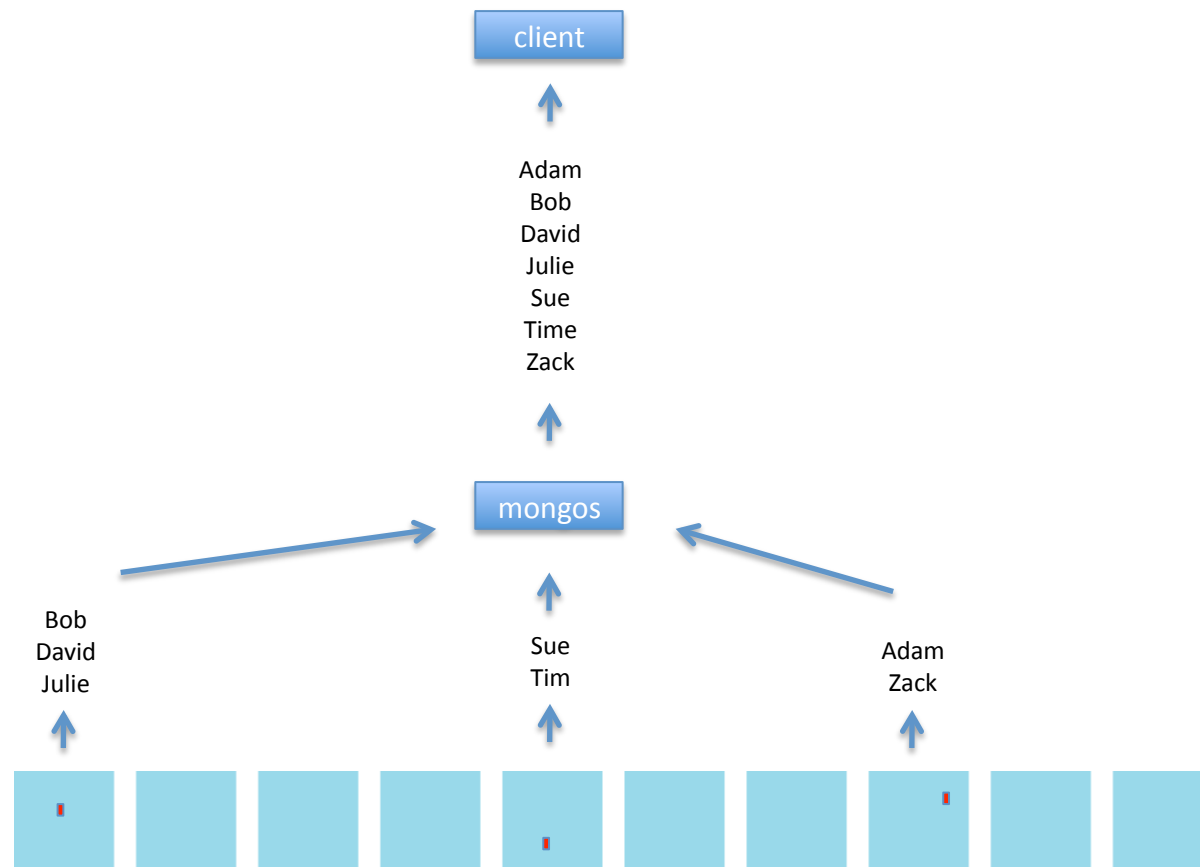
$\{ \underline{a} : \dots, b : \dots, c : \dots \}$ *Non-shard key query, secondary index*

`find({ b : 99, a : 100 })`

Range	Shard
a in $[-\infty, 2000)$	2
a in $[2000, 2100)$	8
a in $[2100, 5500)$	3
...	...
a in $[88700, \infty)$	0

The a term involves the shard key and allows mongos to intelligently route the query to shard 2. Once the query reaches shard 2, the $\{ b : 1 \}$ index can be used to efficiently process the query.

When sorting is specified, the relevant shards sort locally, and then mongos merges the results. Thus the mongos resource usage is not terribly high.



When using replication (typically a replica set), we simply have more than one node per shard.

(arrows below indicate replication, traditional vs. sharded environments)

