MongoDB®

Sharding

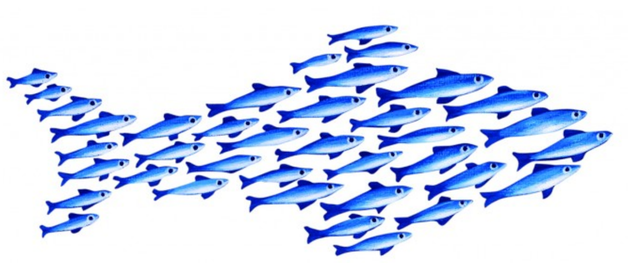
OF300 - Production System Design

Release: 20220512

Topics we cover

What is Sharding

- A mechanism for **partitioning** larger collections across multiple servers
  - Enables bringing unlimited hardware to a scaling problem.
  - Allows parallel processing for some tasks.
  - Allows enforcement of geographical data location.



Sharding is utilized for horizontal scaling.

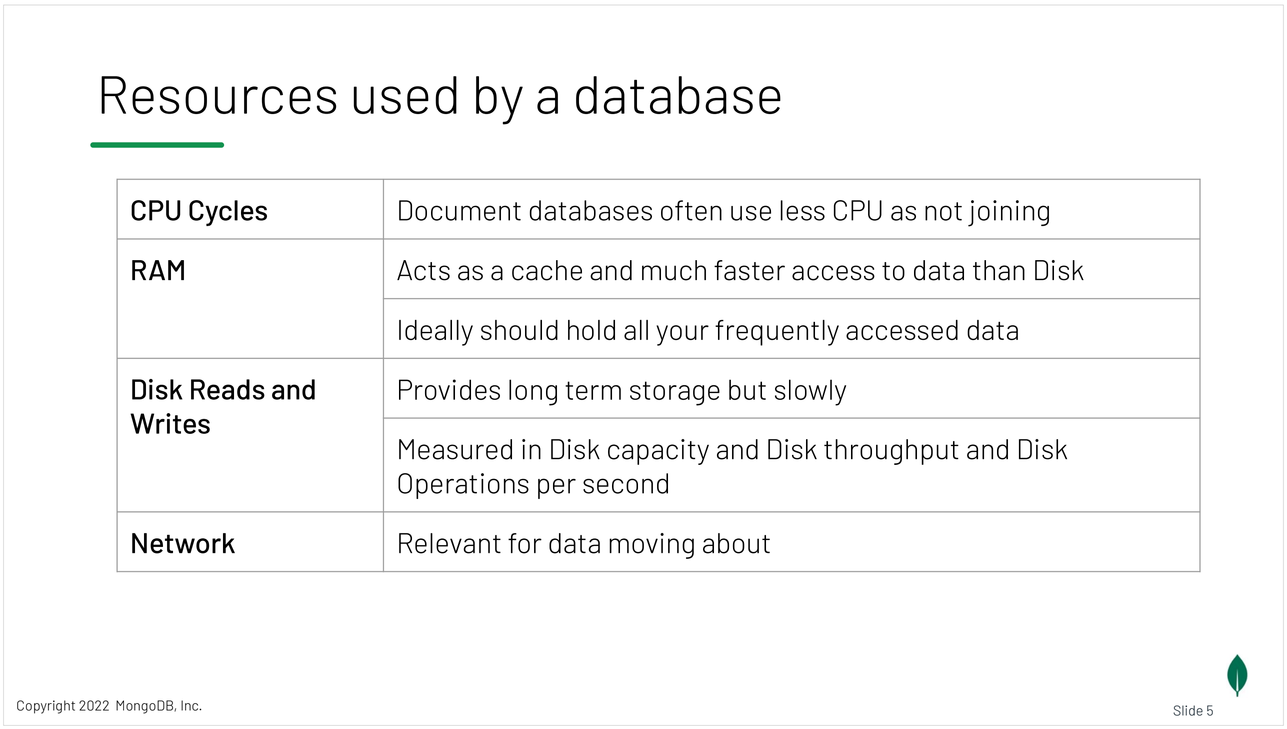
Sharding is not Replication

Replication is to ensure resilience.

Sharding is about bringing more capability.

Many baby ducks is resilience, some don't survive (HA) . Wolves form a pack of adults because a group is more capable than one, they divide up the task of hunting between them working together.

- In some NoSQL systems Scaling and High Availability (HA) use the same mechanism - In MongoDB, they do not
- Replication is used for High Availability
- Sharding is for Scaling Out large workloads.
- Sharding without HA is wrong
  - Increased risk of something breaking not reduced
  - Higher cost and time to repair



Resources used by a database

CPU Cycles	Document databases often use less CPU as not joining
RAM	Acts as a cache and much faster access to data than Disk
	Ideally should hold all your frequently accessed data
Disk Reads and Writes	Provides long term storage but slowly
	Measured in Disk capacity and Disk throughput and Disk Operations per second
Network	Relevant for data moving about

It is essential to understand how the database uses different resources.
What resource is it that is limiting performance, what one will run out first?

Vertical versus Horizontal Scaling

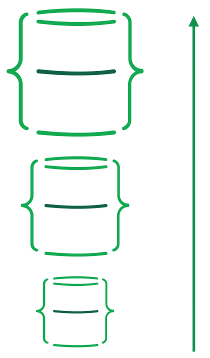
There is a scaling limit
for a single server.

With horizontal scaling,
there is no limit.

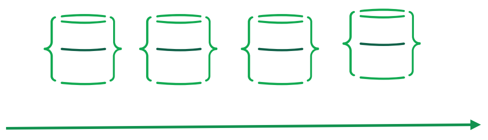
900 Birmingham NEC & Airport | Coventry
SOME PEOPLE A
every 15 mins
national express West Midlands

Vertical versus Horizontal Scaling

Vertical Scaling
Increase size of instances
(RAM, CPU etc)



Horizontal Scaling
Add more instances

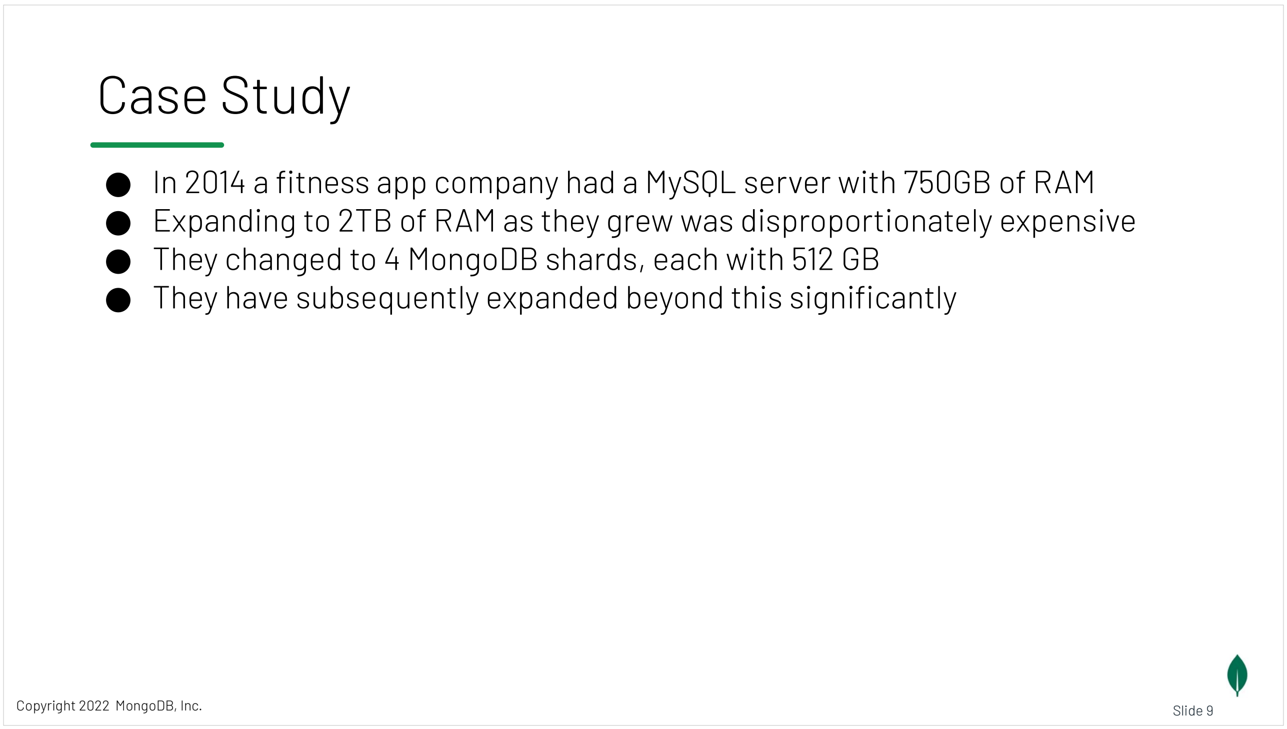


Limitations of Vertical Scaling

Adding more hardware is called Vertical Scaling - it has limits.

- There are limits to how much you can increase a machine’s resources
  - Architecture limits - Max number of cores/RAM
  - Provider Limits - Cloud instance sizes
  - Cost limits - Cost isn't linear once things get large

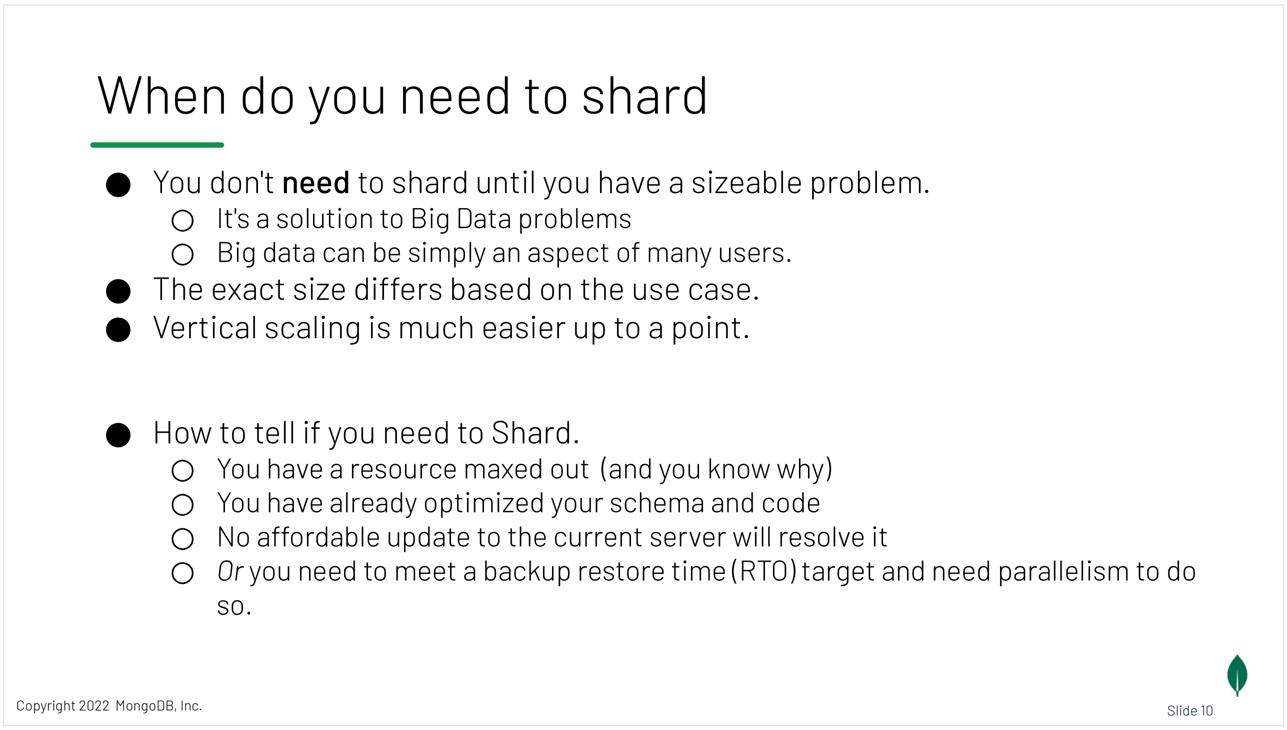
- Vertical scaling has limits that often prevent the ability to scale easily
- Few places are happy to provision a server with 128 cores and 4TB of RAM



Case Study

- In 2014 a fitness app company had a MySQL server with 750GB of RAM
- Expanding to 2TB of RAM as they grew was disproportionately expensive
- They changed to 4 MongoDB shards, each with 512 GB
- They have subsequently expanded beyond this significantly

Example of how application growth can take advantage of horizontal scaling

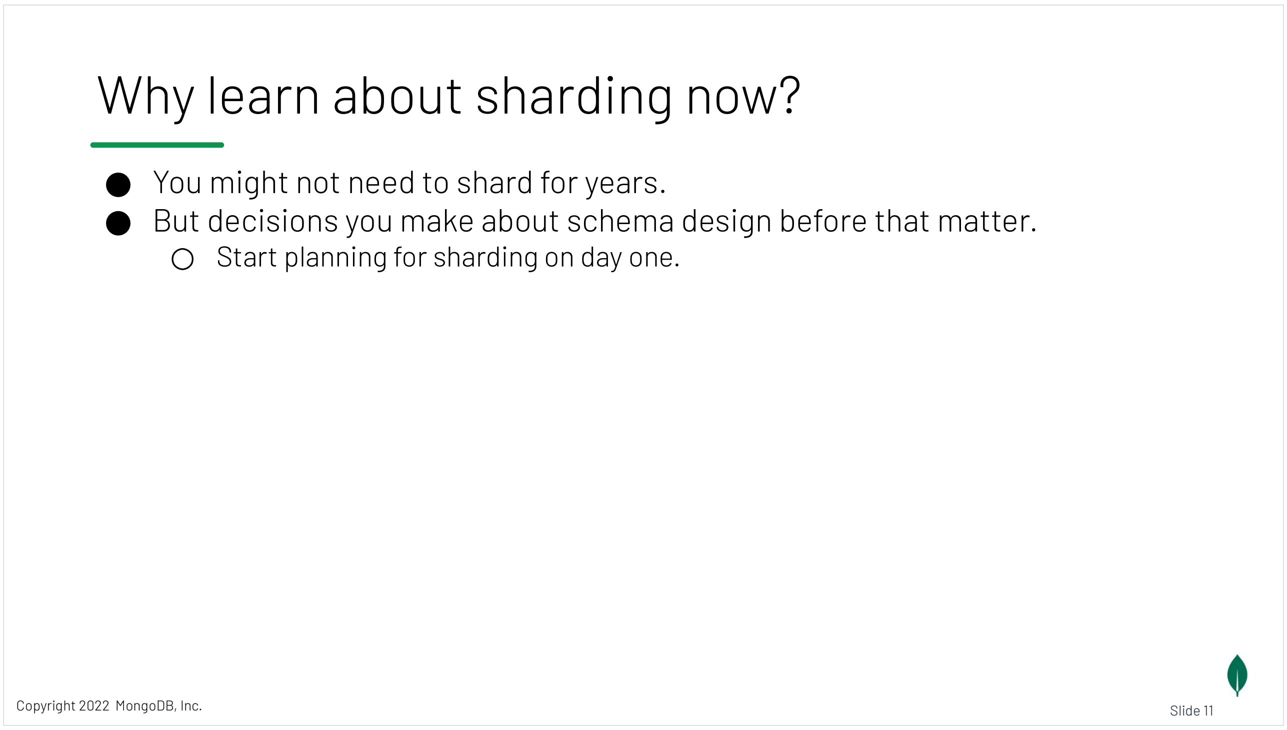


When do you need to shard

- You don't **need** to shard until you have a sizeable problem.
  - It's a solution to Big Data problems
  - Big data can be simply an aspect of many users.
- The exact size differs based on the use case.
- Vertical scaling is much easier up to a point.

- How to tell if you need to Shard.
  - You have a resource maxed out (and you know why)
  - You have already optimized your schema and code
  - No affordable update to the current server will resolve it
  - *Or* you need to meet a backup restore time (RTO) target and need parallelism to do so.

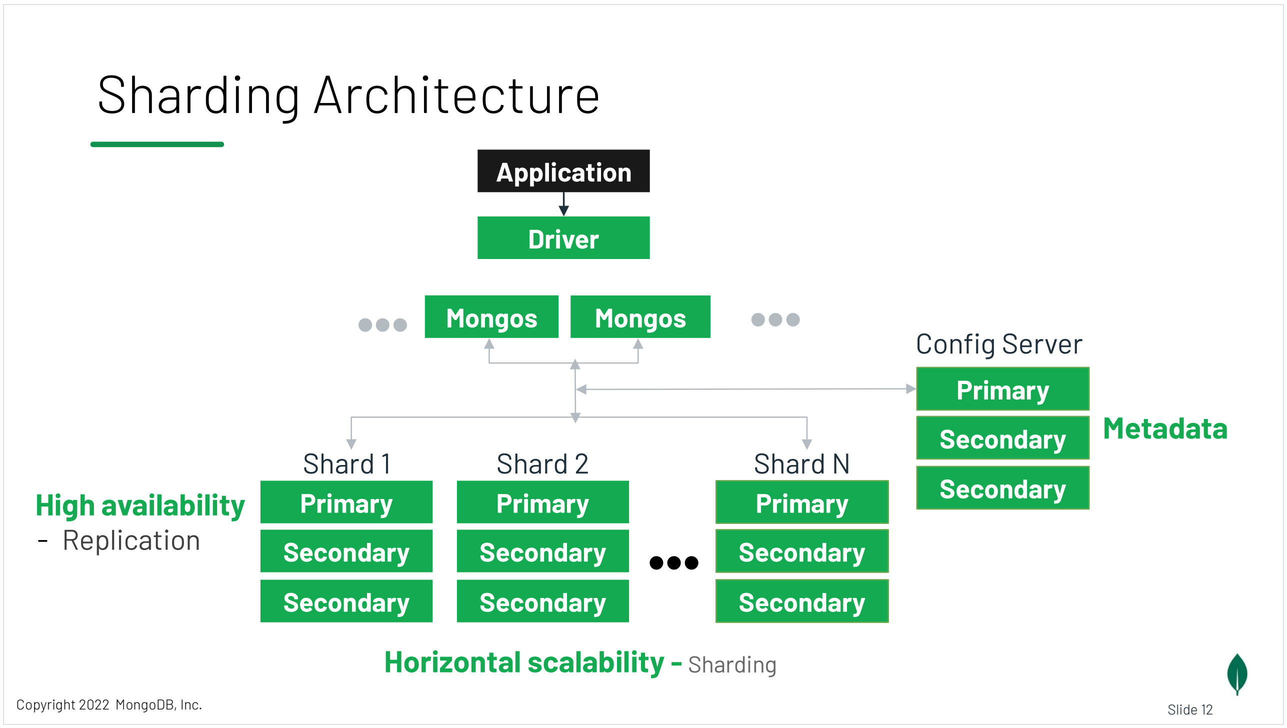
Sharding is not required until your data grows to a 2TB maximum, but sharding before that - maybe 200GB is recommended sometimes.



Why learn about sharding now?

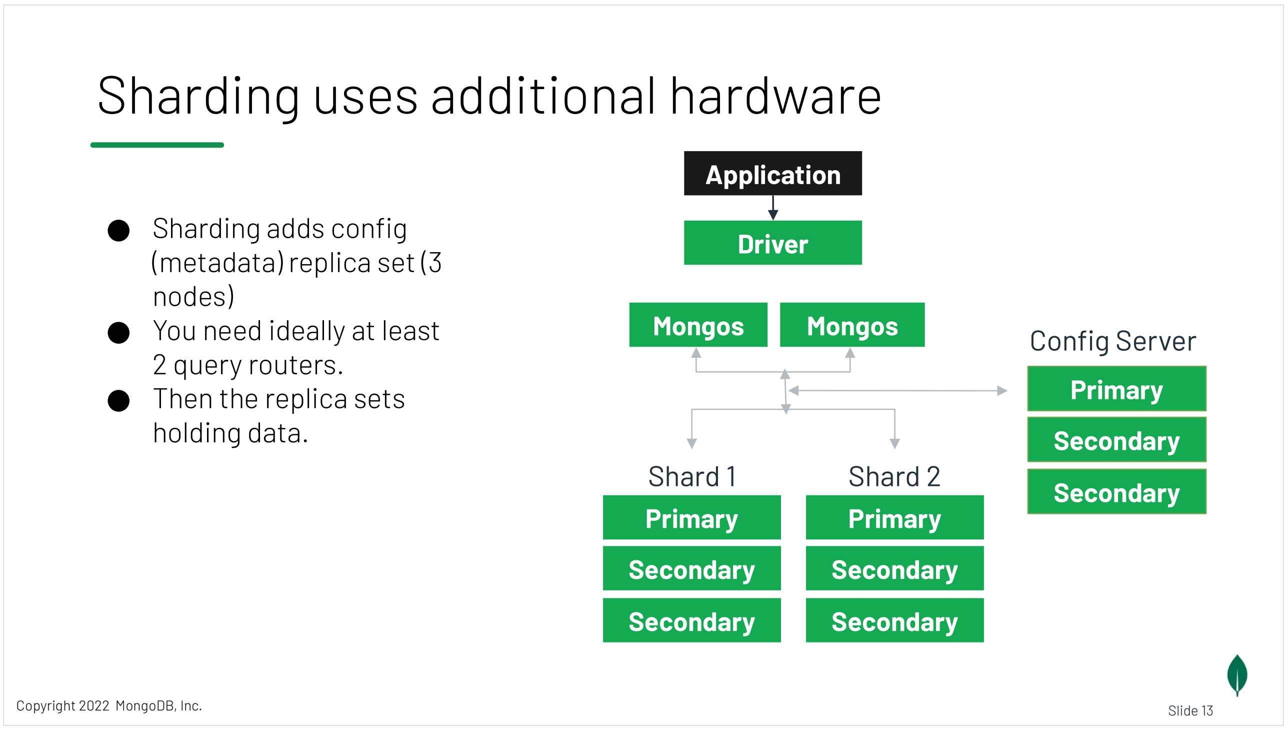
- You might not need to shard for years.
- But decisions you make about schema design before that matter.
    - Start planning for sharding on day one.

It is important to understand the concept of sharding so that you can plan for future growth.



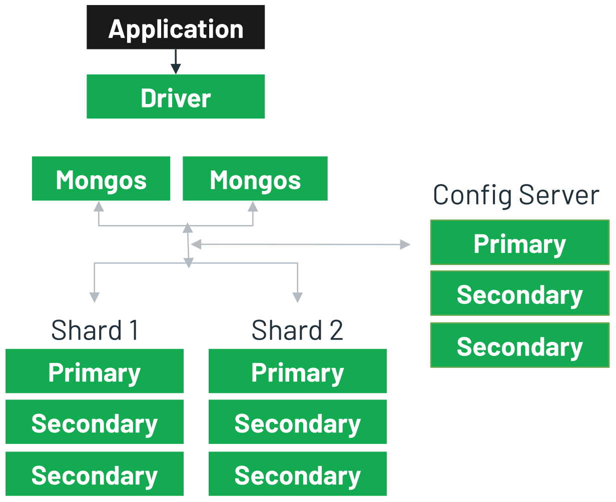
Sharding Architecture

Sharding has a number of moving parts - each of which should be planned and sized. Sharding is about solving big problems - you can do a huge amount of processing before you need to go there so don't be afraid of the complexity - remember that you may be managing a 200 server cluster ultimately and that means you are doing something that has real world value.

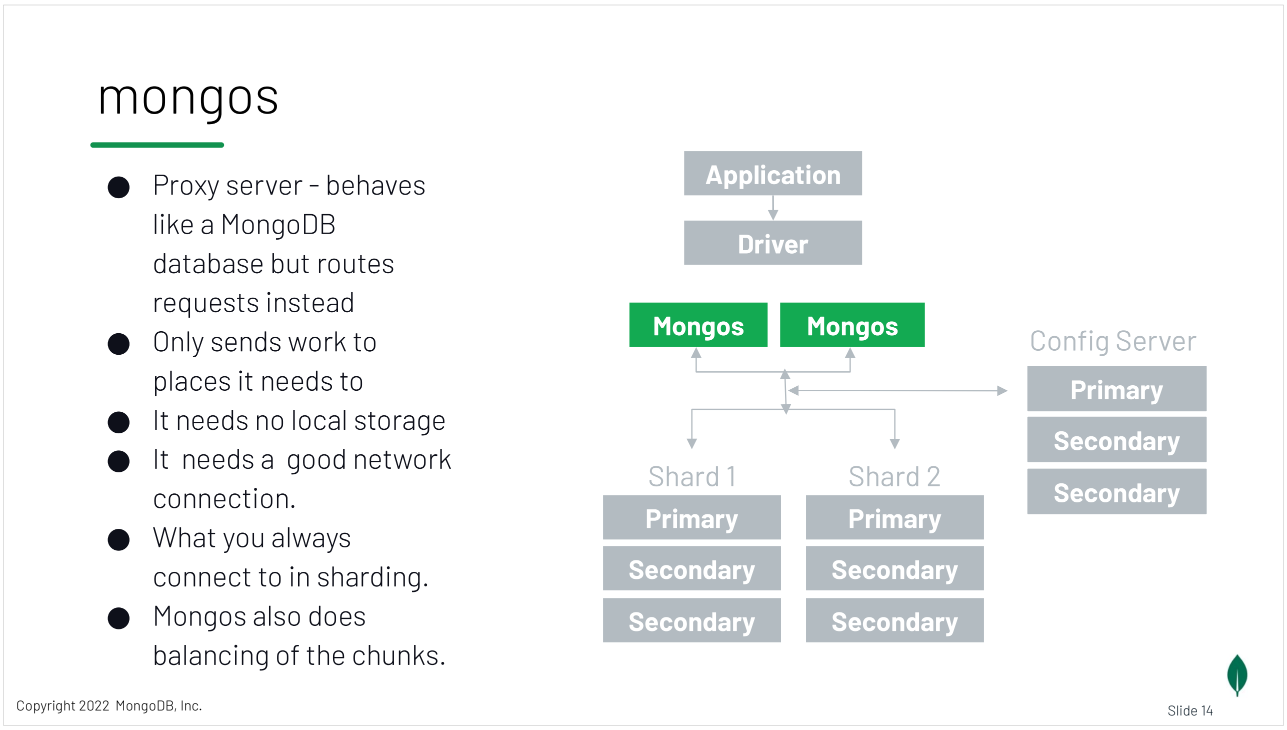


Sharding uses additional hardware

- Sharding adds config (metadata) replica set (3 nodes)
- You need ideally at least 2 query routers.
- Then the replica sets holding data.

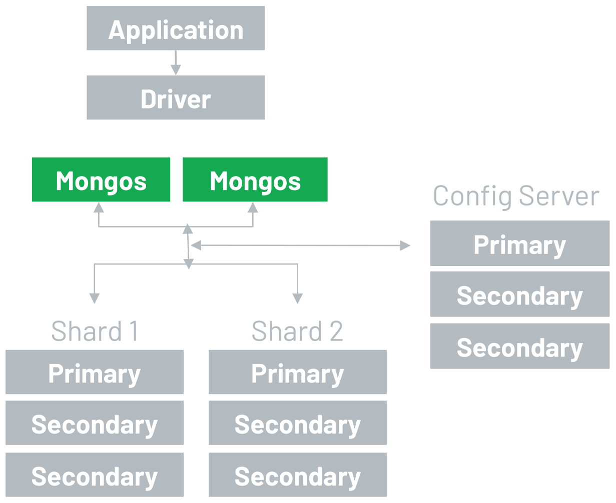


- The smallest production cluster should be 11 nodes.
- 2 Replica Sets to hold data
- 1 Replica set to hold metadata
- 2 Routers to avoid a single point of failure
- Sometimes these are combined - e.g., a router on a data server.



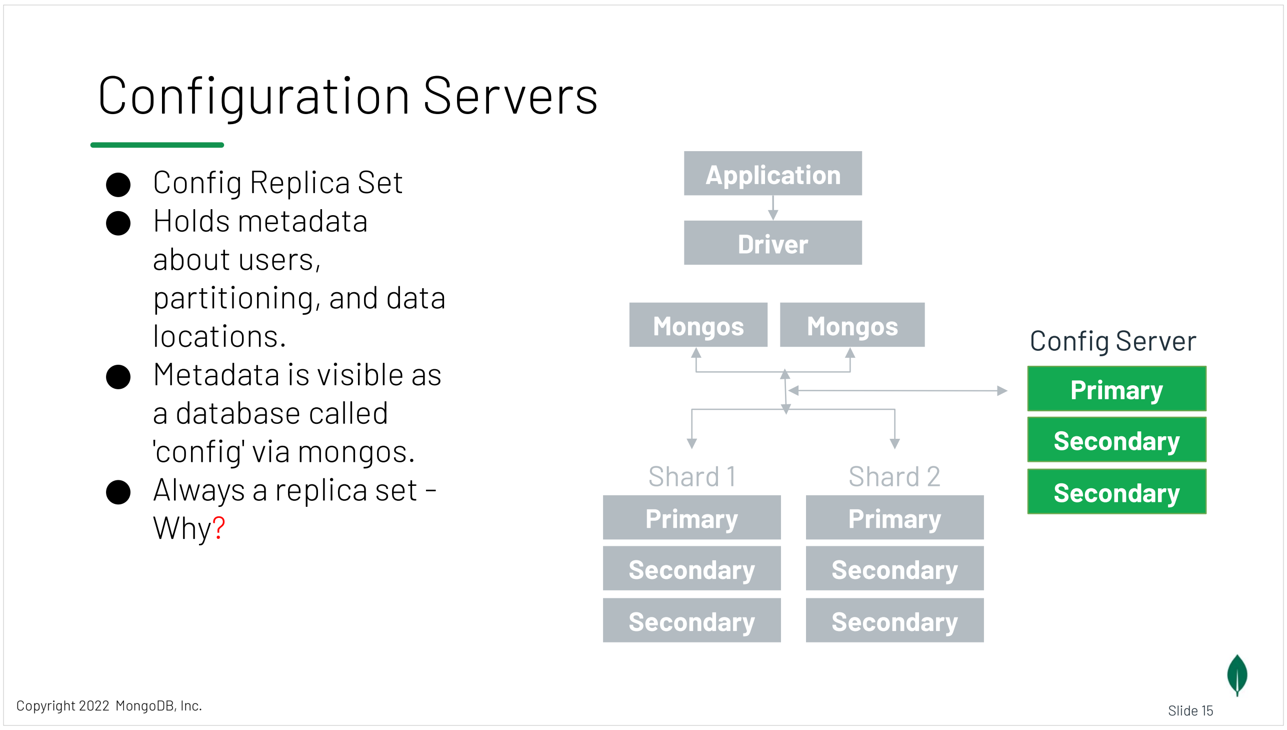
mongos

- Proxy server - behaves like a MongoDB database but routes requests instead
- Only sends work to places it needs to
- It needs no local storage
- It needs a good network connection.
- What you always connect to in sharding.
- Mongos also does balancing of the chunks.



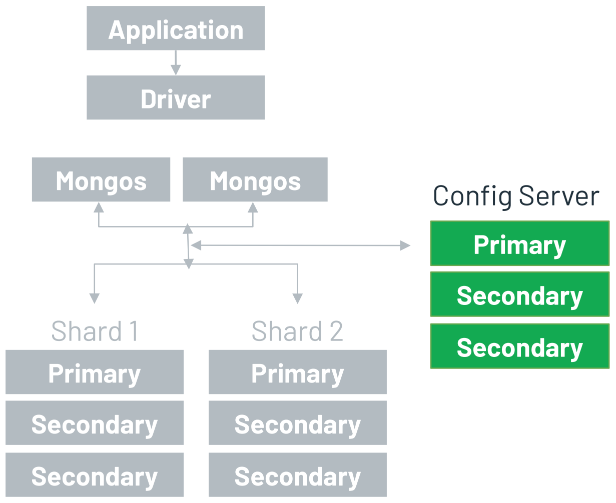
mongos routes requests (replaces mongod from standalone / replica set)

mongos will balance the chunks by monitoring chunk distribution across different shards and coordinate the chunk migration.

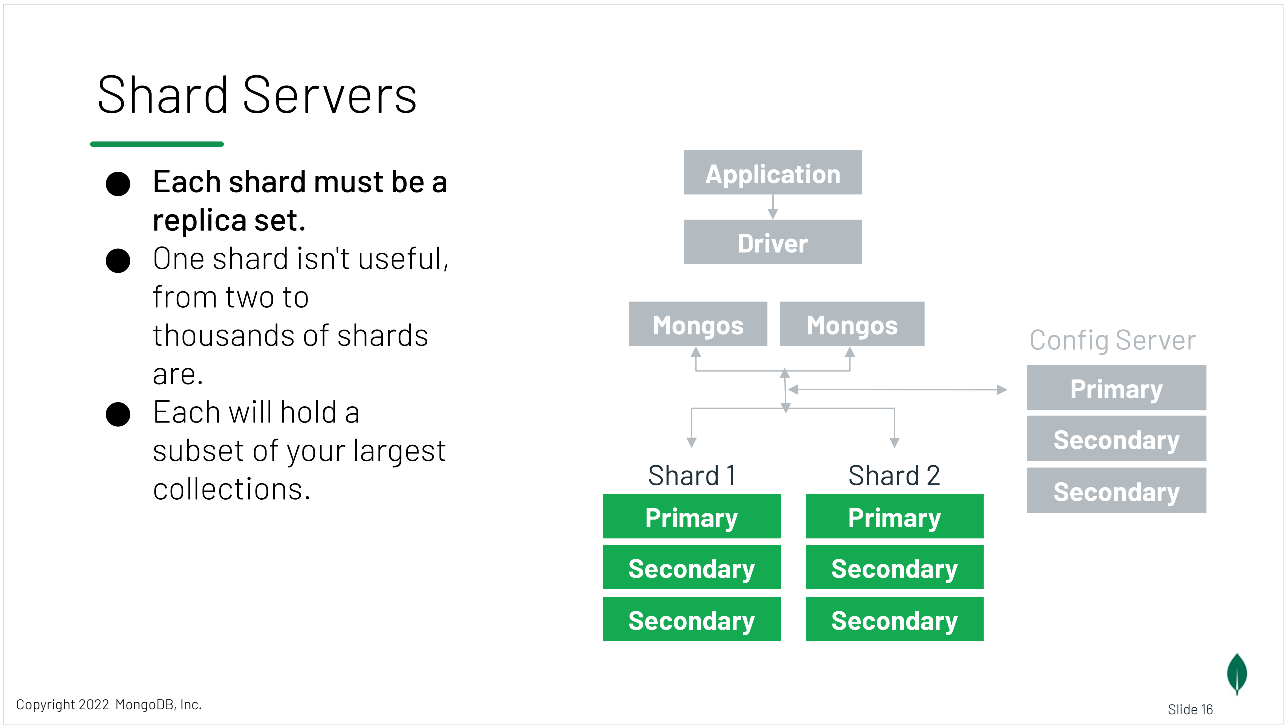


Configuration Servers

- Config Replica Set
- Holds metadata about users, partitioning, and data locations.
- Metadata is visible as a database called 'config' via mongos.
- Always a replica set - Why?

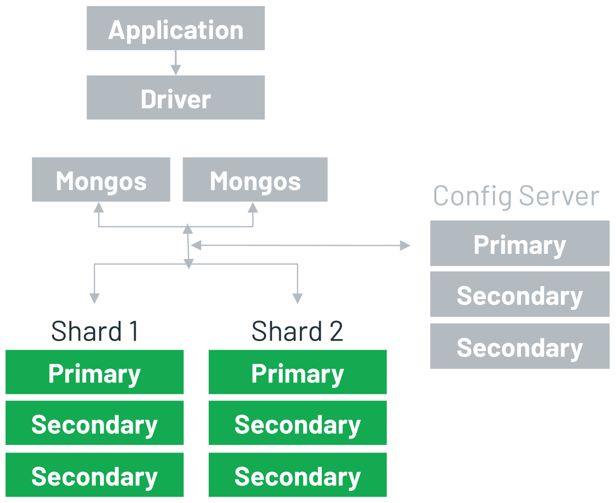


Configuration servers hold metadata about users, partitioning, location of ranges of data..

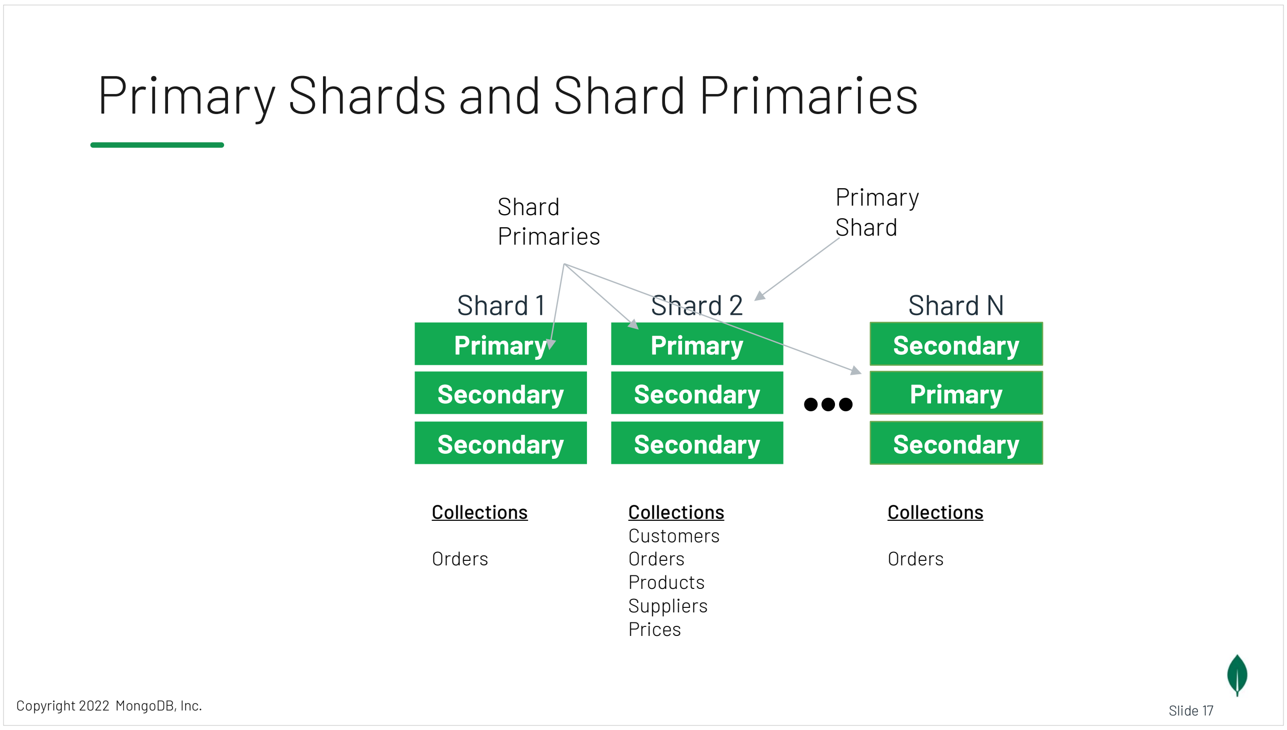


Shard Servers

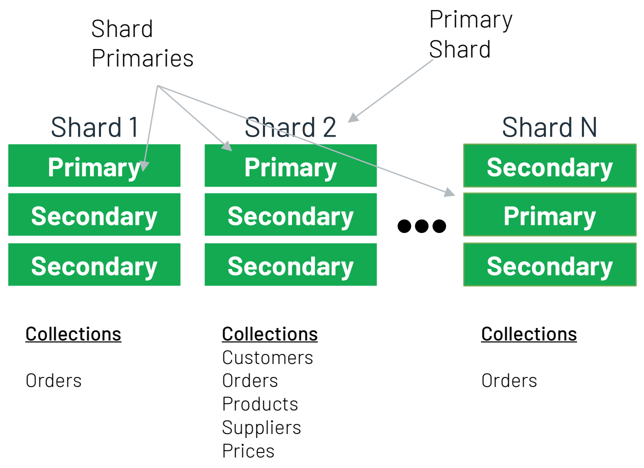
- **Each shard must be a replica set.**
- One shard isn't useful, from two to thousands of shards are.
- Each will hold a subset of your largest collections.



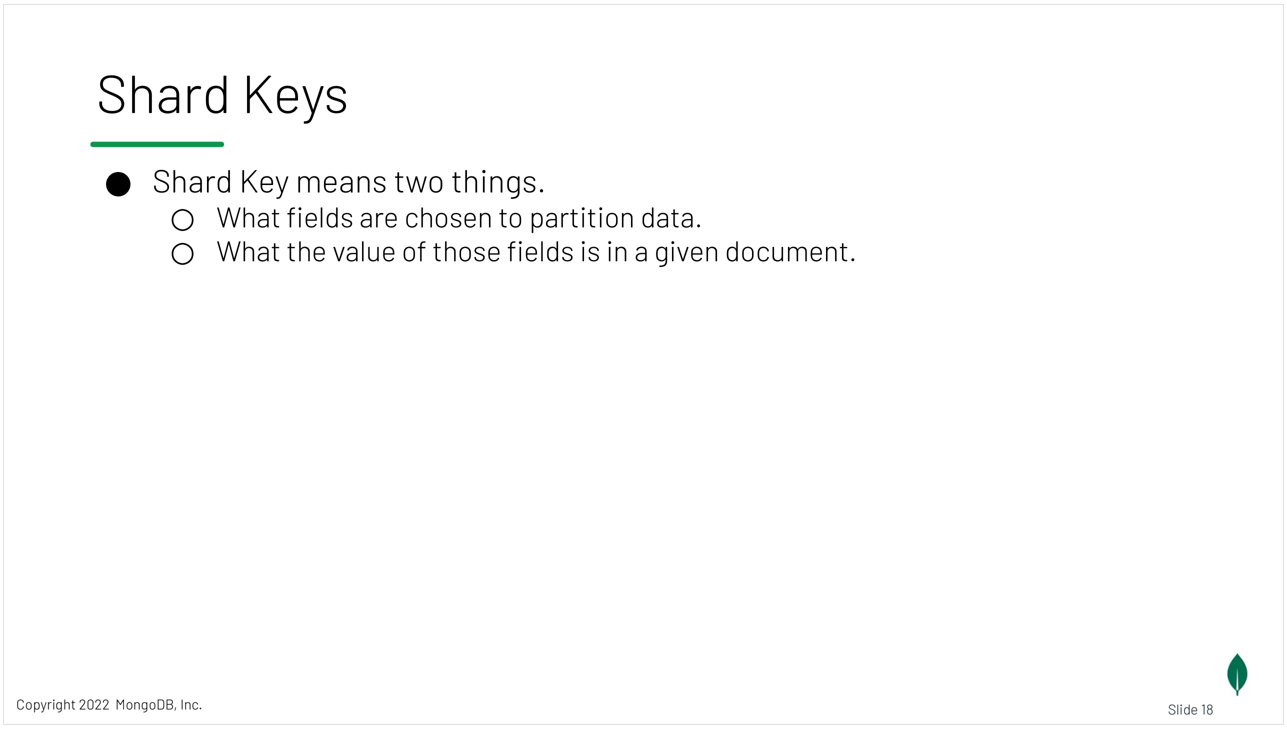
- Each shard is a replica set.
- We do not want a shard to be unavailable; hence each shard must be a replica set.
- Each additional server statistically increases the chance of any one server failing, by using replica set, a server failing doesn't break the system.



Primary Shards and Shard Primaries



A Shard Primary is whatever node is the current Primary in the Replica Set representing that Shard.
The Confusingly similar term "Primary Shard" is the shard , for each database, designated when sharding was enabled to hold any unsharded collections in that database,
The shard primary is also the shard used for any other tasks that cannot be done across all the shards.

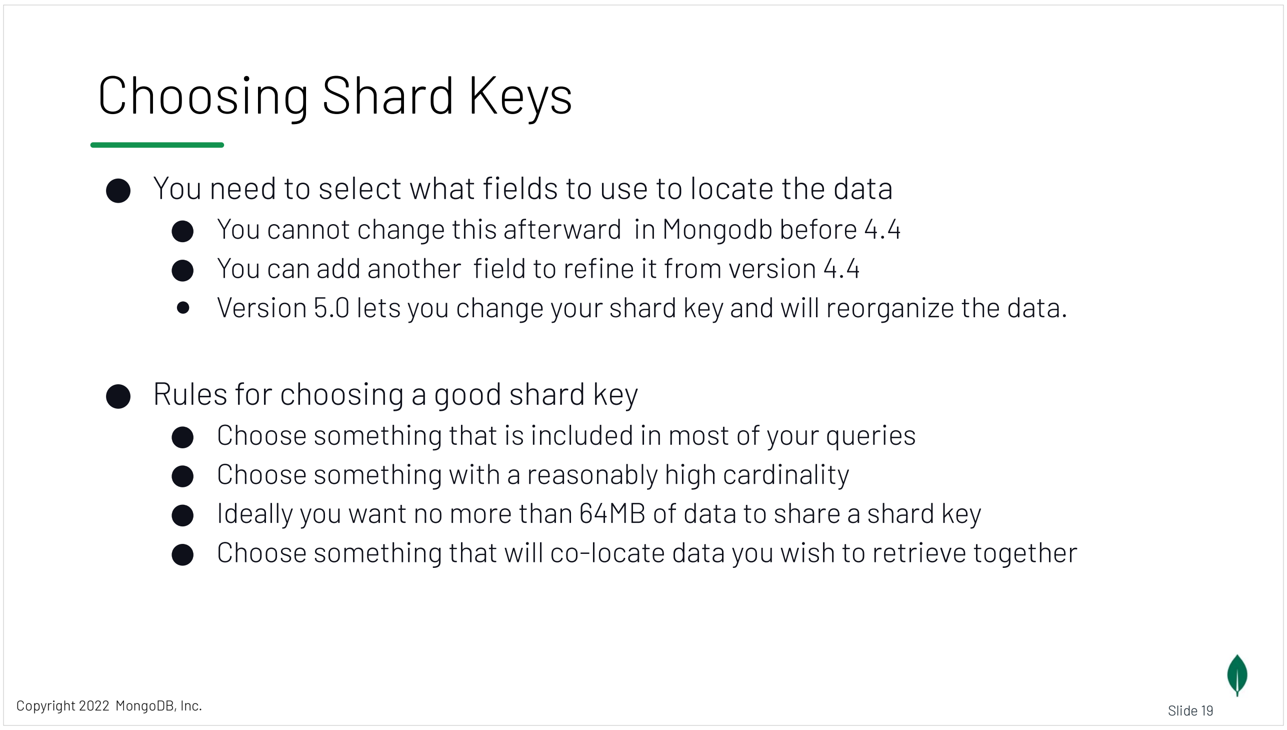


Shard Keys

- Shard Key means two things.
  - What fields are chosen to partition data.
  - What the value of those fields is in a given document.

The term Shard key is used in two different but related ways.
The shard key of a collection is what the field names and order are for the partitioning.
For example, partition (shard) it by surname and county.

But in a document, the shard key is Jones and Orange - the value of the shard key.



Choosing Shard Keys

- You need to select what fields to use to locate the data
  - You cannot change this afterward in Mongodb before 4.4
  - You can add another field to refine it from version 4.4
  - Version 5.0 lets you change your shard key and will reorganize the data.

- Rules for choosing a good shard key
  - Choose something that is included in most of your queries
  - Choose something with a reasonably high cardinality
  - Ideally you want no more than 64MB of data to share a shard key
  - Choose something that will co-locate data you wish to retrieve together

A shard key will determine how your data is partitioned.
If the cardinality is too low or too many items have the same value then you will get 'Jumbo" chunks - which cannot be split and balanced.

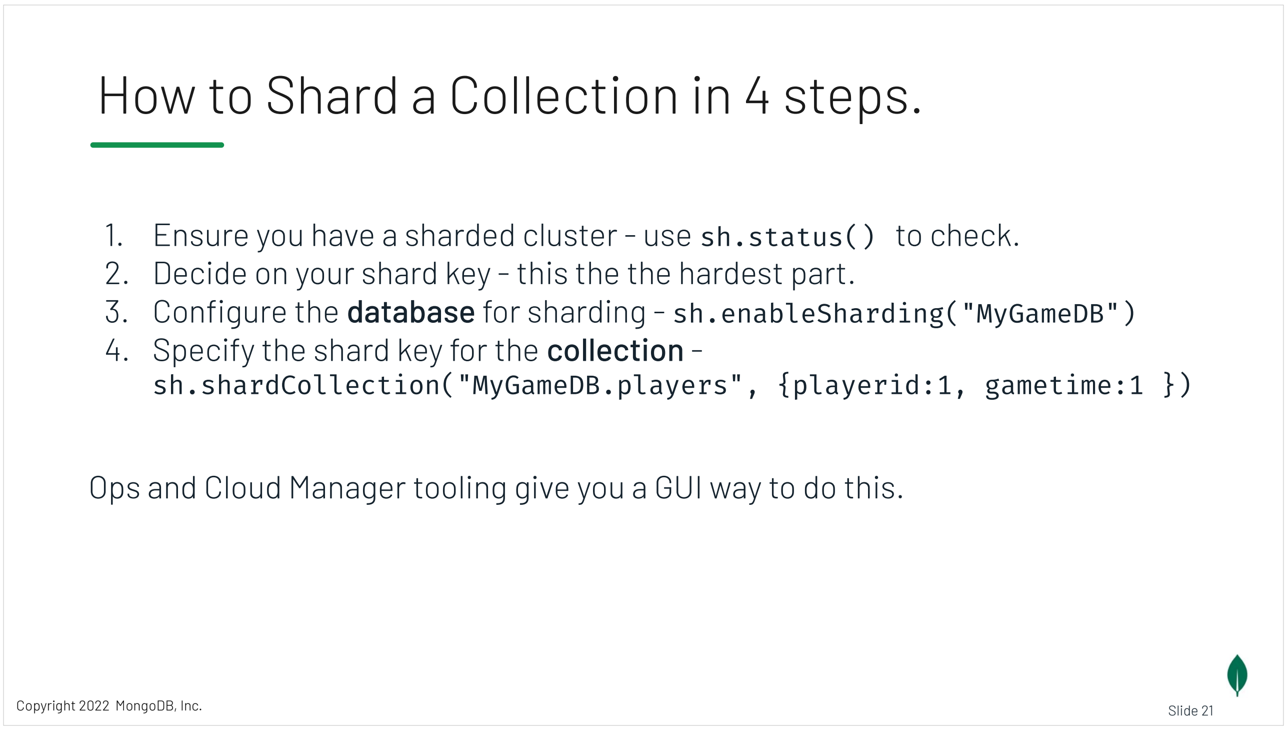
Choosing shard keys in practice

Typically, it isn't challenging to choose a shard key; however, many who do so give it no thought and then have problems later.

- You only shard large collections.
- In large collections
  - Users work with obvious subsets of the data
  - If it's 'their' data - shard by a user, for example, bank accounts or games
  - If it's departmental - shard by department or branch
- Where there are no clear subsets
  - Shard for parallelism
  - For instance, in an analytic data store, add a random value for the shard key.

The choice of a shard key should be considered carefully! It cannot be changed without dropping and reloading your data.

Like a ship in a bottle - once it's done, it's hard to undo; equally - unless someone teaches you how, you might wonder how best to put the ship in the bottle in the first place.

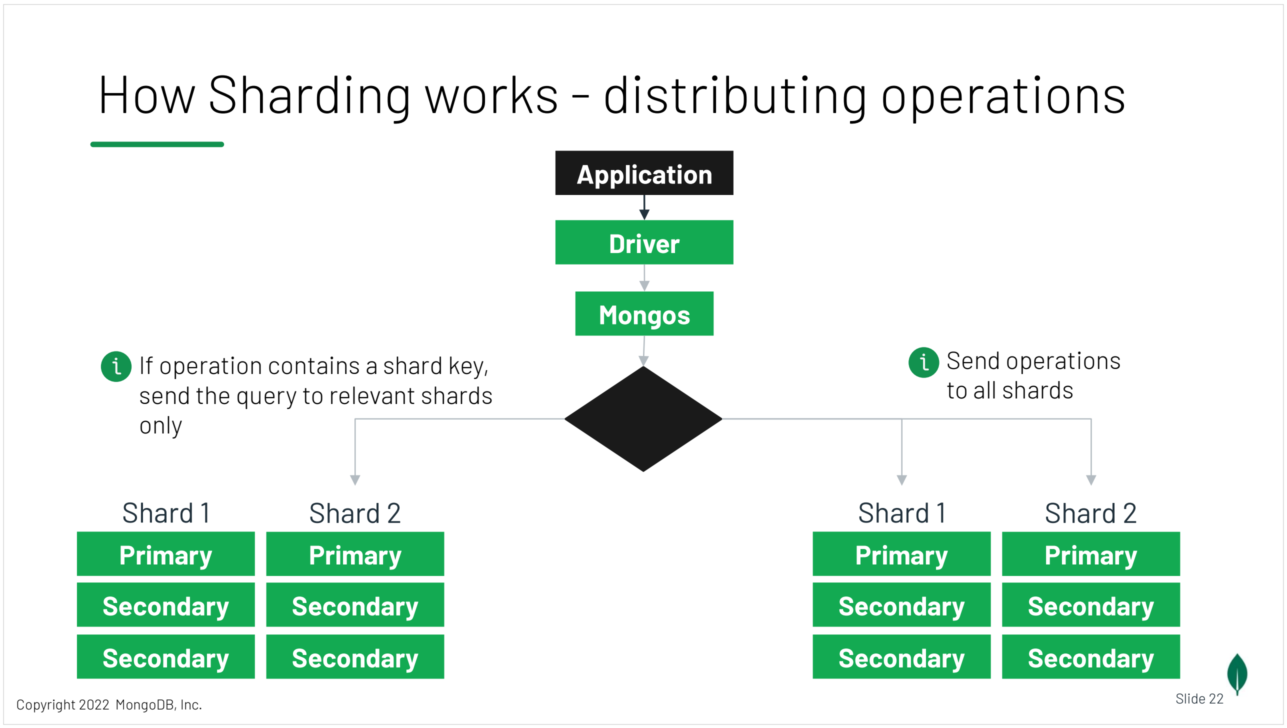


How to Shard a Collection in 4 steps.

1. Ensure you have a sharded cluster - use `sh.status()` to check.
2. Decide on your shard key - this the the hardest part.
3. Configure the **database** for sharding - `sh.enableSharding("MyGameDB")`
4. Specify the shard key for the **collection** -
   `sh.shardCollection("MyGameDB.players", {playerid:1, gametime:1 })`

Ops and Cloud Manager tooling give you a GUI way to do this.

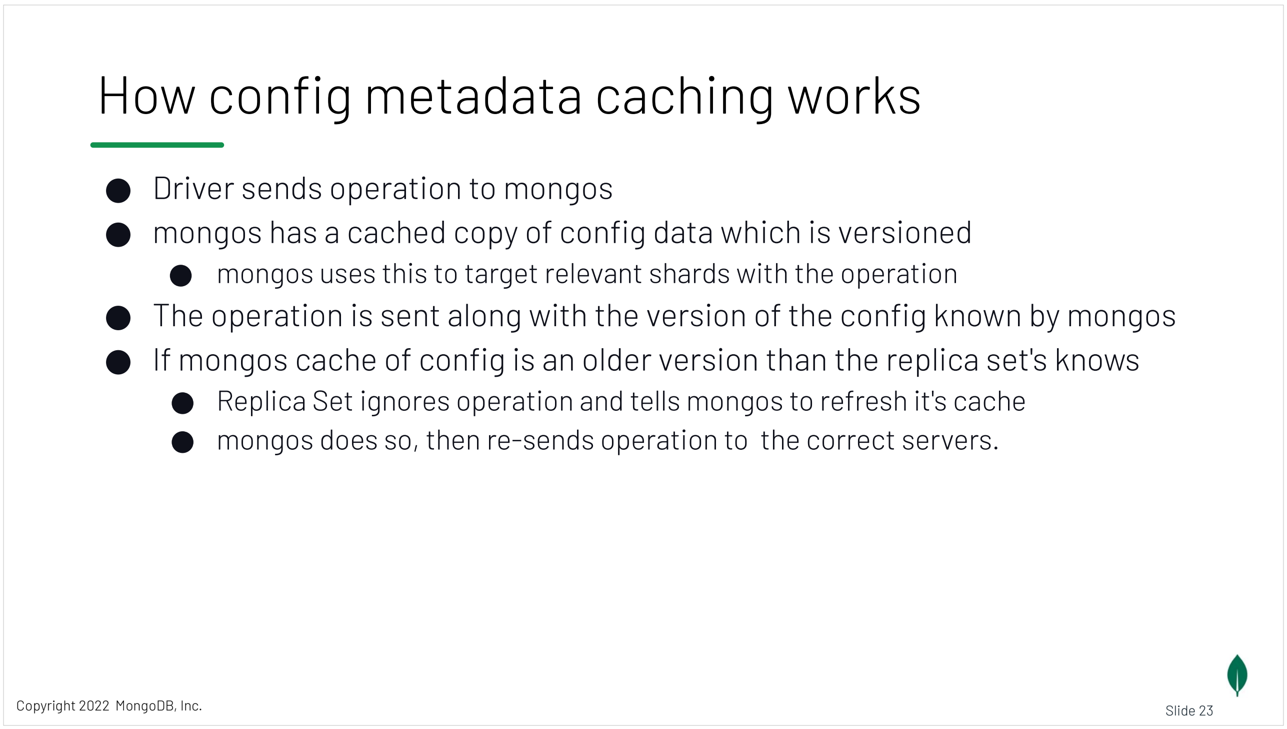
If your collection already exists and has data then you will also need to create an index which beings with (or is) the shard key before you can run shardCollection. MongoDB needs this index to efficiently find and move chunks.



How Sharding works - distributing operations

- The query is sent to mongos by the driver
- IF query contains shard key (or at least first field of the key)
  - Lookup where data with that key is stored
  - Send query to those replica sets to run as normal
  - When results come back stream to the client.

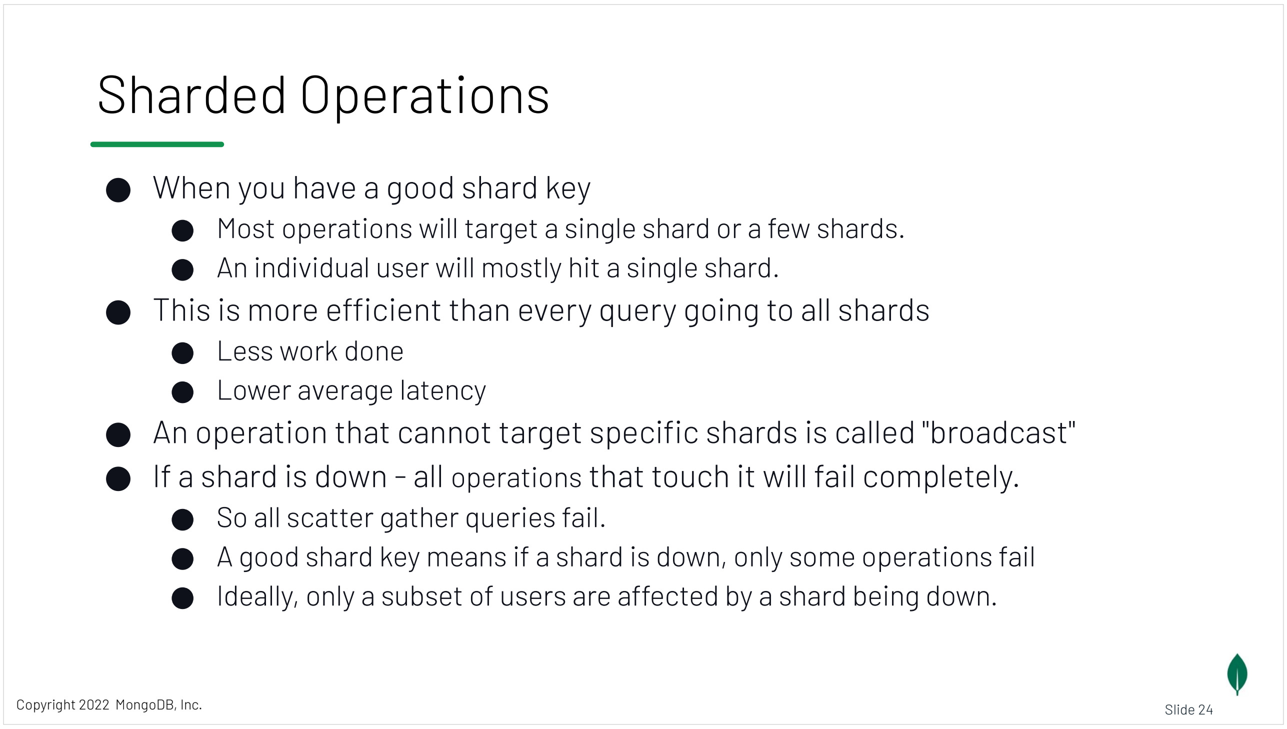
  ELSE
  - Send query to all servers.
  - When results come back stream to the client.
- Concepts regarding reading from secondaries and read concerns still apply and the sharded replica sets are queried appropriately.
- If sorting required, do merge-sort of results as they come back



How config metadata caching works

- Driver sends operation to mongos
- mongos has a cached copy of config data which is versioned
  - mongos uses this to target relevant shards with the operation
- The operation is sent along with the version of the config known by mongos
- If mongos cache of config is an older version than the replica set's knows
  - Replica Set ignores operation and tells mongos to refresh it's cache
  - mongos does so, then re-sends operation to the correct servers.

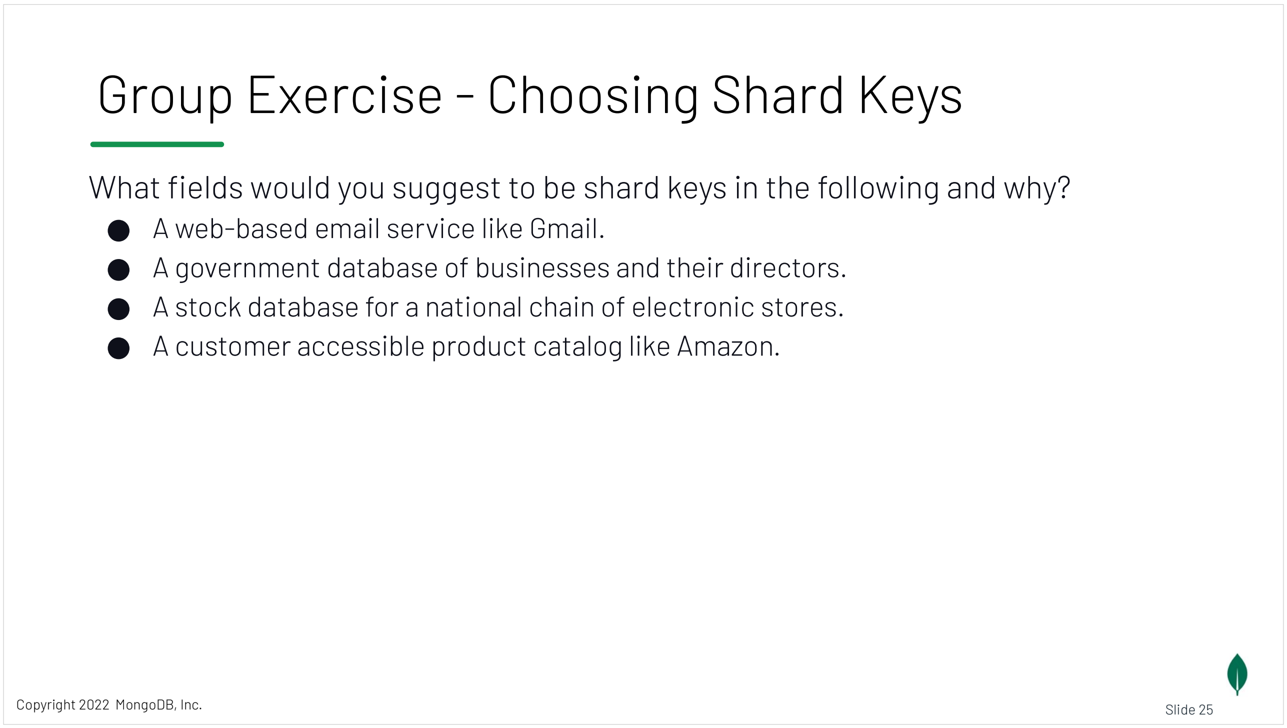
mongos doesn't need to ask the config servers every time they have a config 'version' which they send to the replica set with the write (or read) - the replica set know the most up to date config version and will tell the mongos if it needs to refresh its metadata.



Sharded Operations

- When you have a good shard key
  - Most operations will target a single shard or a few shards.
  - An individual user will mostly hit a single shard.
- This is more efficient than every query going to all shards
  - Less work done
  - Lower average latency
- An operation that cannot target specific shards is called "broadcast"
- If a shard is down - all operations that touch it will fail completely.
  - So all scatter gather queries fail.
  - A good shard key means if a shard is down, only some operations fail
  - Ideally, only a subset of users are affected by a shard being down.

A good shard key will target few shards, preventing a “scatter gather”



Group Exercise - Choosing Shard Keys

What fields would you suggest to be shard keys in the following and why?

- A web-based email service like Gmail.
- A government database of businesses and their directors.
- A stock database for a national chain of electronic stores.
- A customer accessible product catalog like Amazon.

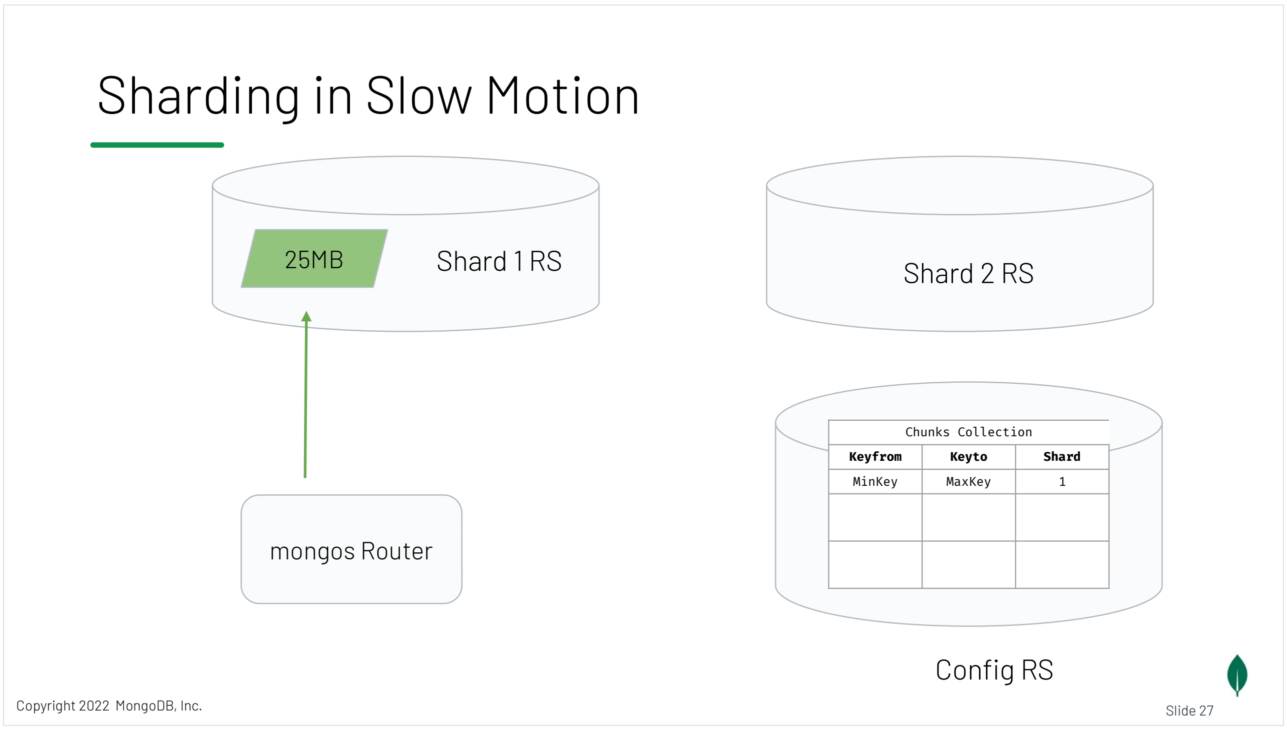
Exercise

Chunks

- A Chunk is a term used to refer to all documents where the shard key is in a given range of values.
- Each Chunk exists on a single shard.
- If a shard key falls into the range, then it is set to be in the Chunk.
- Data in each range is always inclusive of its lower boundary and exclusive of its upper boundary.
- The reads/writes are routed to a shard hosting the specific Chunk.

Sharded collection data is stored in chunks.

For example, Customer number 667726 might be in a chunk going from 660000 to 670000, and that chunk is stored on shard 8. Inserts, updates, or queries for that value will be sent just to shard 8.

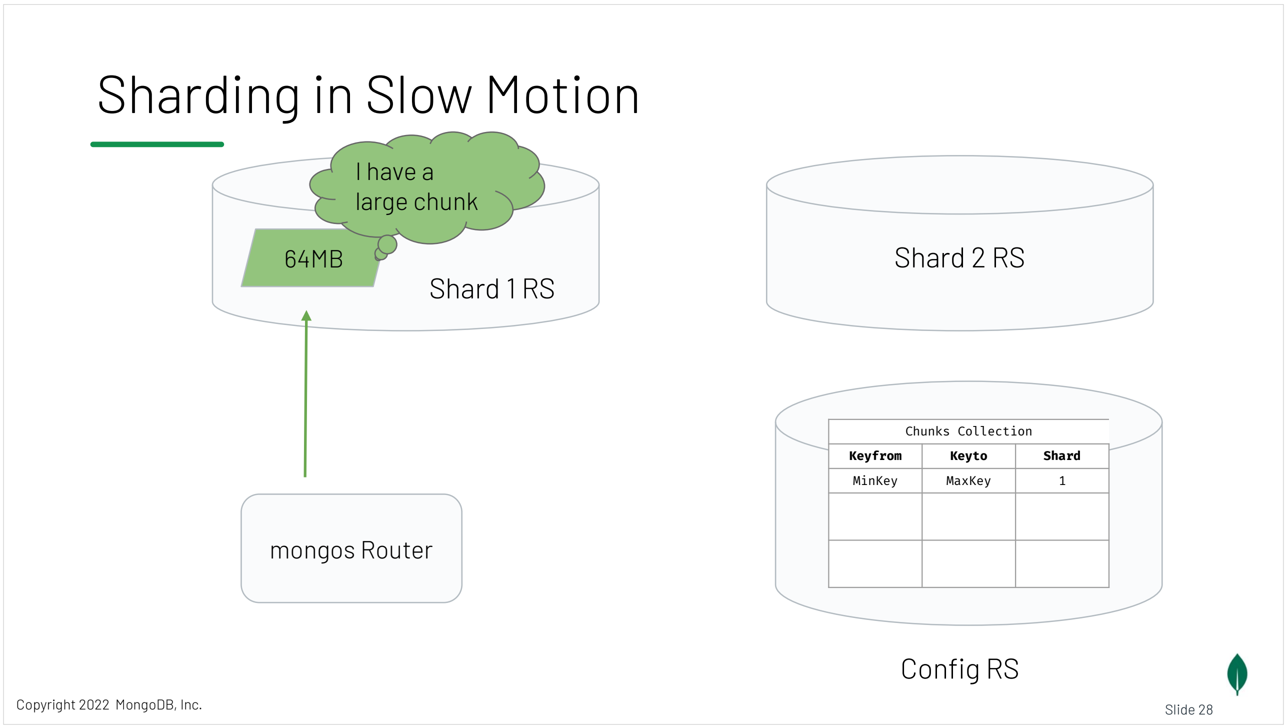


Sharding in Slow Motion

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	MaxKey	1

- With a newly sharded collection, it contains just one chunk from MinKey to MaxKey, which will hold all documents
- Therefore all writes initially go to a single shard.



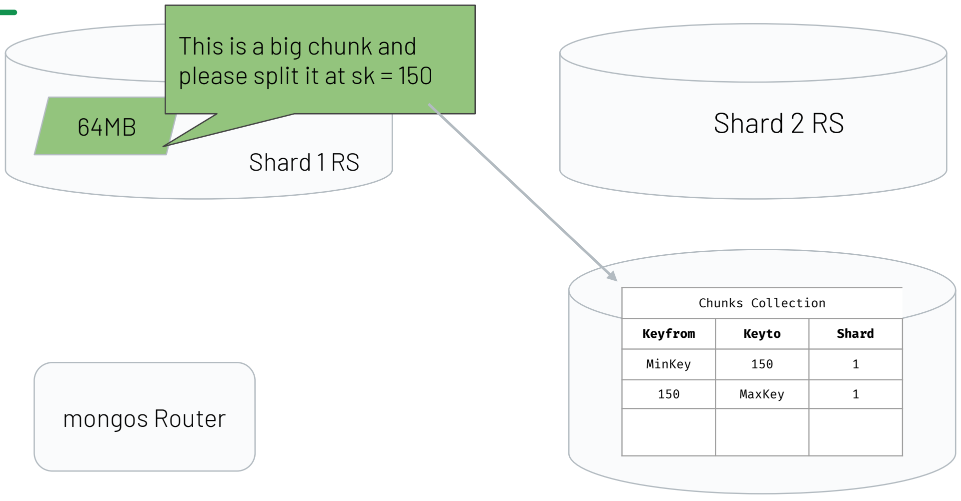
Sharding in Slow Motion

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	MaxKey	1

- The shard will realize at some point that a chunk has just written too is getting large
- We define what this is, but we want chunks to be small enough to move them easily. So by default, we want them to be less than 64MB

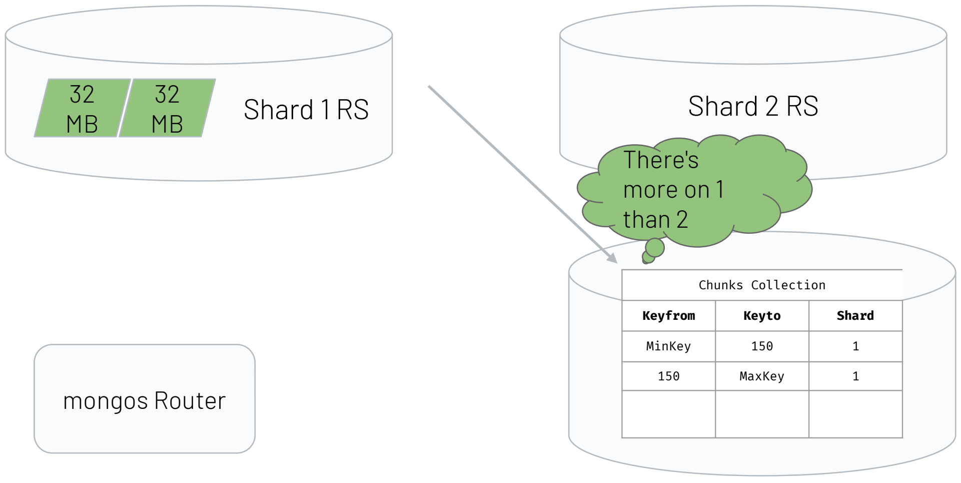
Sharding in Slow Motion



Keyfrom	Keyto	Shard
MinKey	150	1
150	MaxKey	1

- The shard will work out where a good point to split it is and tell the config server this is a good change
- The config server will update its metadata (also, the routers will download this change) - no data moved during this auto splitting process.

Sharding in Slow Motion

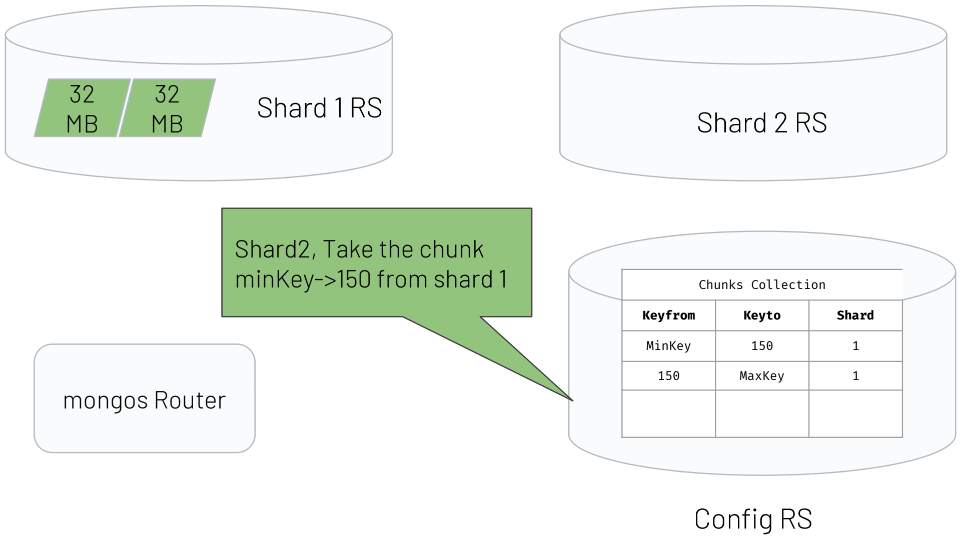


Chunks Collection

Keyfrom	Keyto	Shard
MinKey	150	1
150	MaxKey	1

At this point, some code called the balancer notices all is not right.

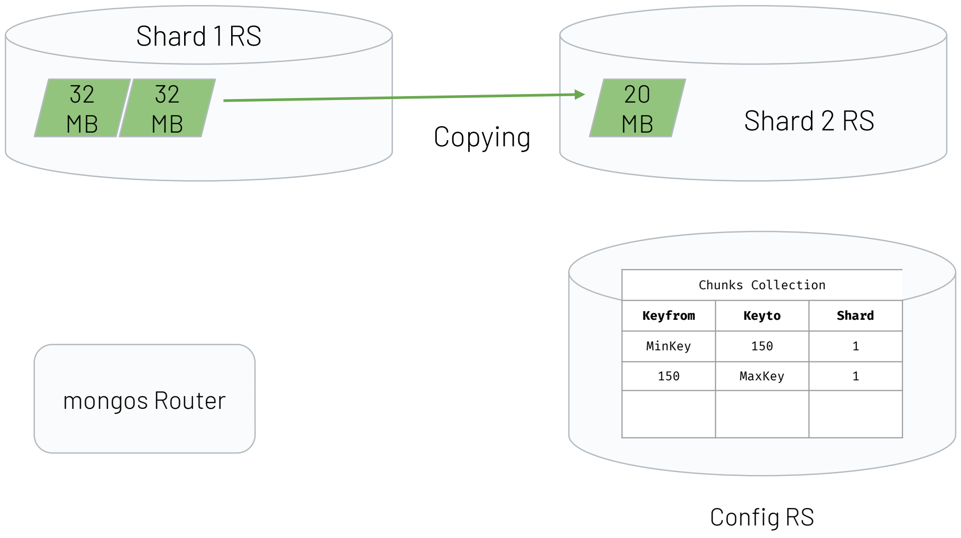
Sharding in Slow Motion



Chunks Collection		
Keyfrom	**Keyto**	**Shard**
MinKey	150	1
150	MaxKey	1

Instruction to initiate moving chunk from Shard 1 to Shard 2.

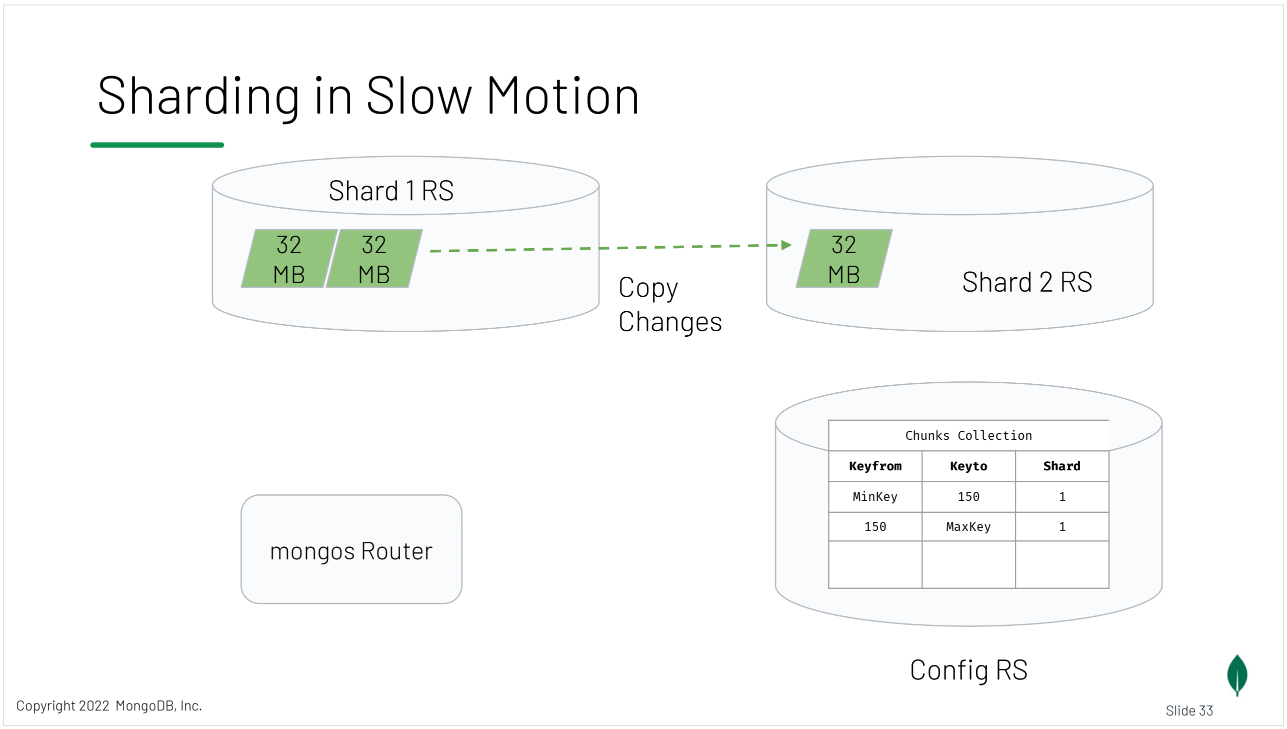
Sharding in Slow Motion



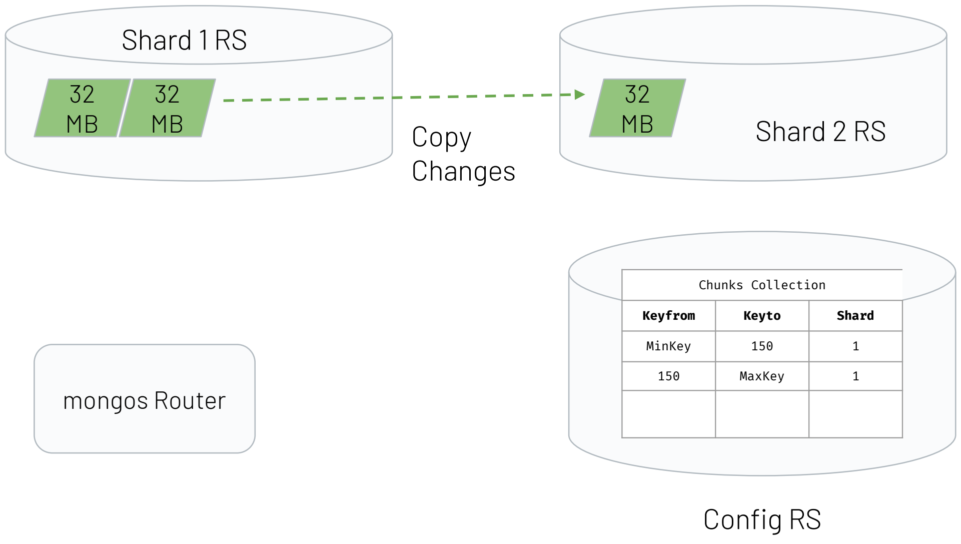
Chunks Collection

Keyfrom	Keyto	Shard
MinKey	150	1
150	MaxKey	1

Shard 2 will begin copying the chunk from Shard 1



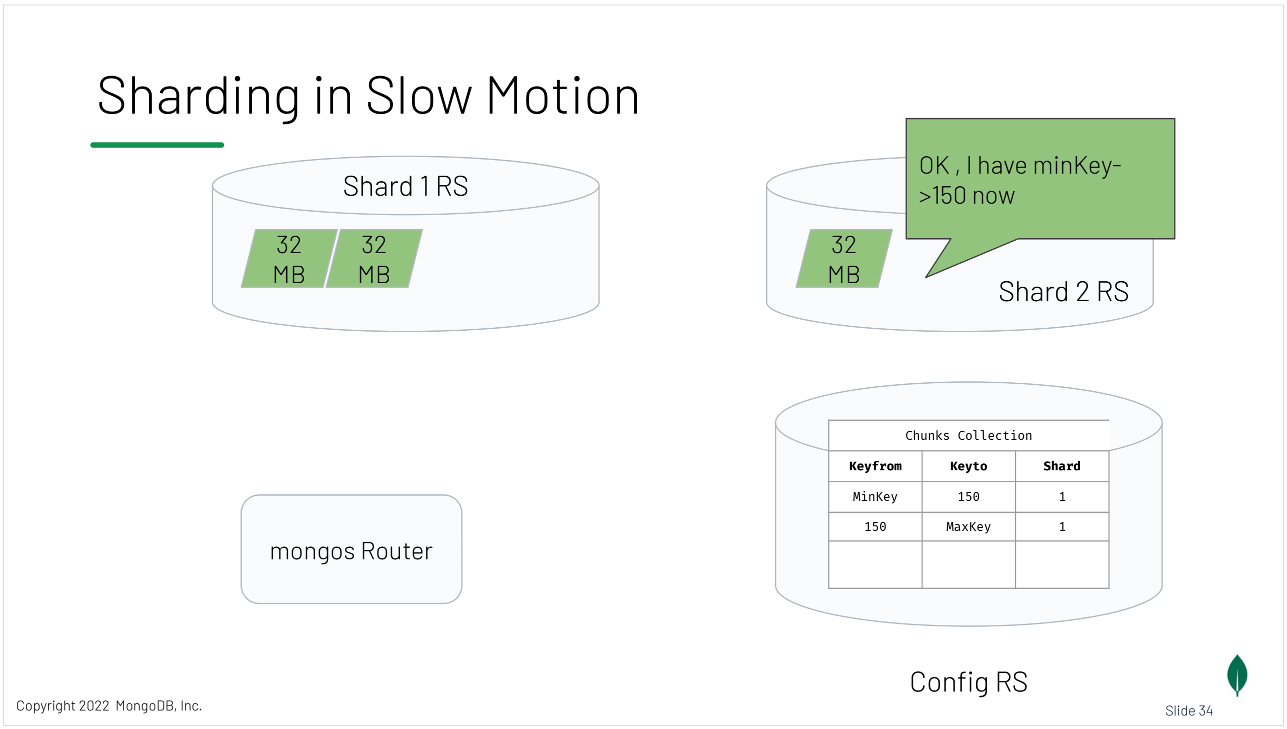
Sharding in Slow Motion



Chunks Collection

Keyfrom	Keyto	Shard
MinKey	150	1
150	MaxKey	1

Shard 2 will eventually have the full chunk from Shard 1



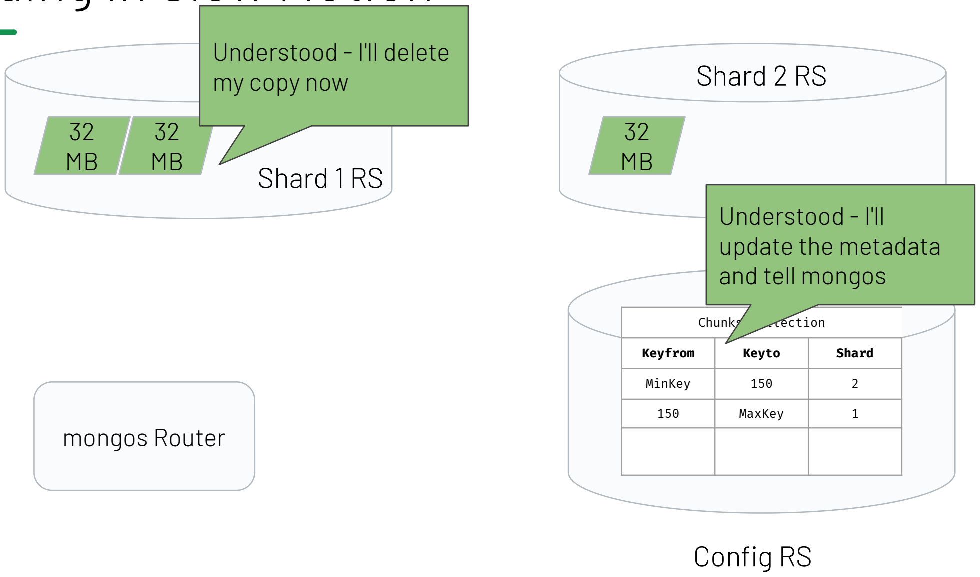
Sharding in Slow Motion

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	150	1
150	MaxKey	1

Shard 2 will confirm it has the full chunk.

Sharding in Slow Motion


Understood - I'll delete my copy now
32 MB
32 MB
Shard 1 RS
Shard 2 RS
32 MB
Understood - I'll update the metadata and tell mongos
mongos Router
Config RS

Chunks collection

Keyfrom	Keyto	Shard
MinKey	150	2
150	MaxKey	1

- Shard 1 will delete the chunk that has been copied
- Config server will update its metadata

Presplitting before loading

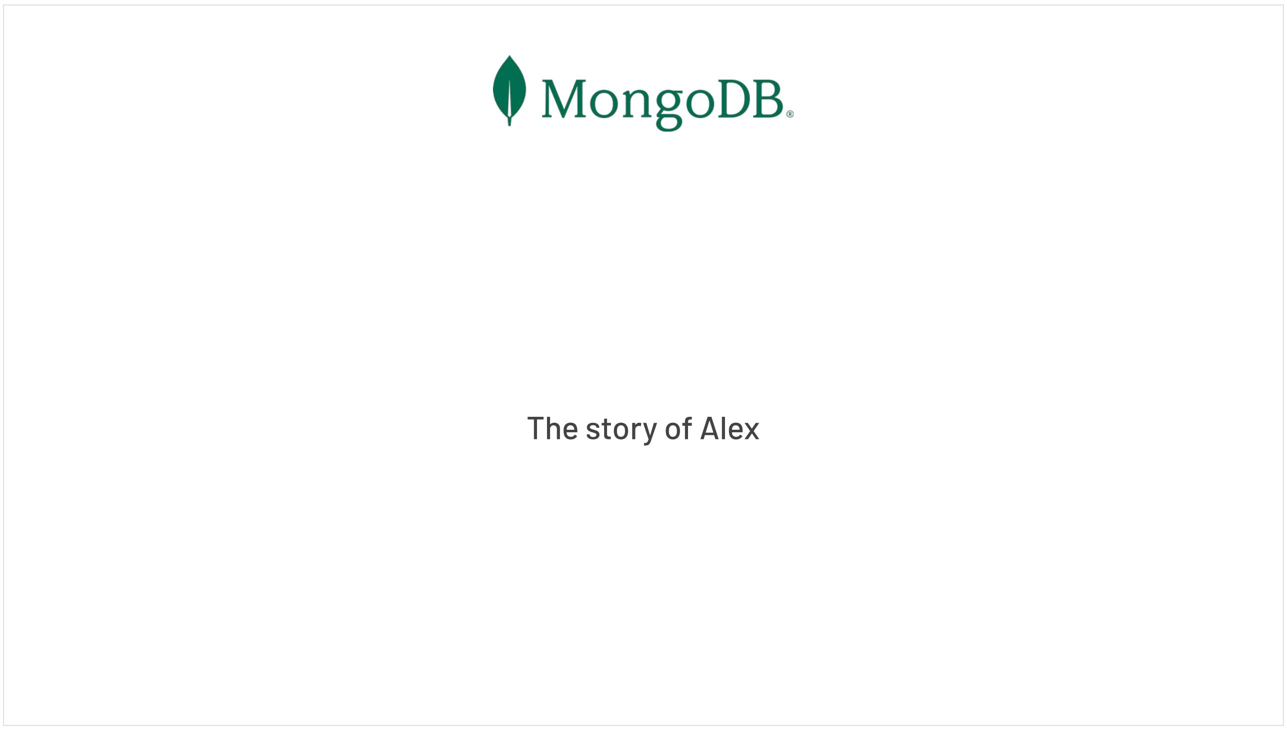
- To avoid the initial shuffling of data via the first shard we can presplit
- We explicitly split the chunk before it has any data in.
- We can then either let the balancer move the empty chunks...
- Or we can explicitly move them which may be faster.
- Doing this correctly can greatly speed up a bulk load of data.

One thing we can do is, before loading any data is manually split the original chunk by specifying where it should be split.
Doing this generally requires us to have an idea of the range of values in the shard key.
Once we have split the original chunk into smaller (still empty - just logical) chunks we can also move them.
Moving an empty chunk takes very little time as there is no data to copy, it's just a metadata change.
This is done with the commands splitChunk and moveChunk
We only need to create one chunk per shard, especially if we move them manually.
A chunk cannot move BACK to a shard it came from for 15 minutes - so be careful when manually moving not to do this - the command hangs for a 15 minute timeout.
Best to split first then move all the chunks and do not move a chunk to the shard it's already on.

Sharding Pitfalls

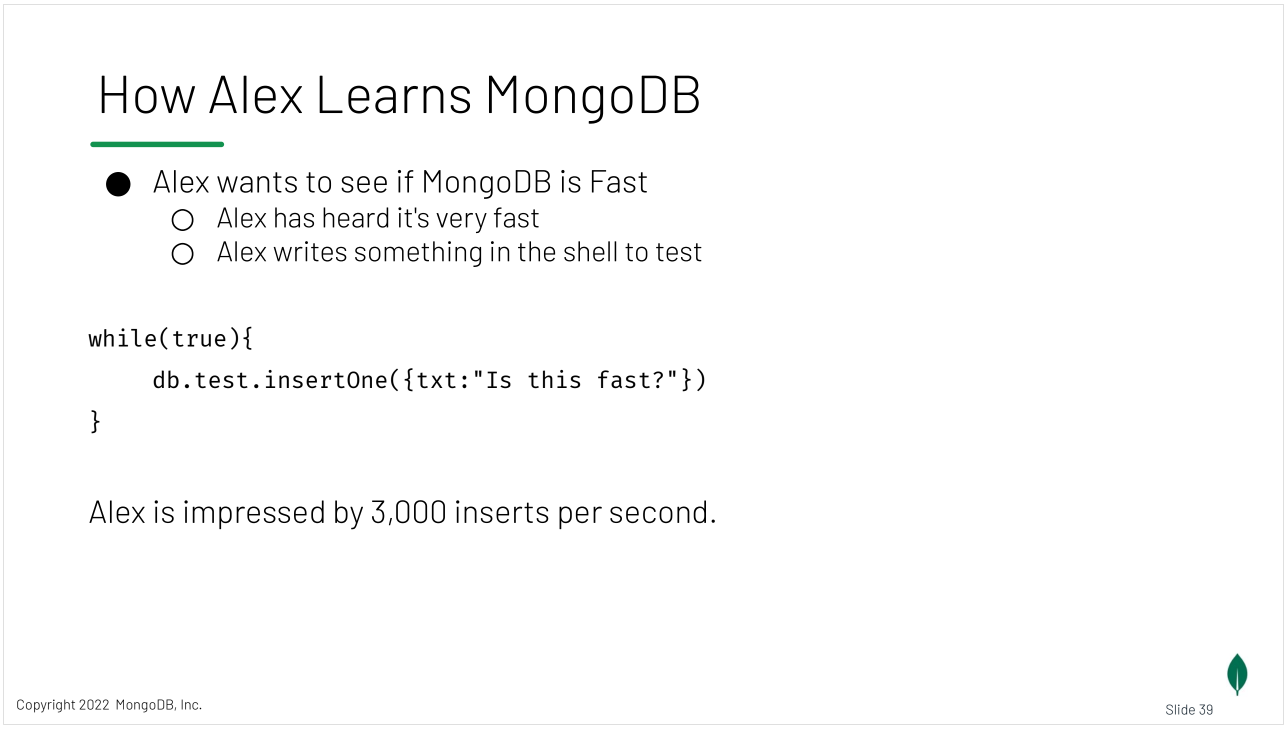
- Having to balance this is not very efficient
  - Each chunk moved is being written three times (write on 1, write on 2, delete on 1)
  - In a busy system getting the deltas can take time (more reads and writes)
  - Once chunks are distributed, as long as the writes hit random chunks, it's good.
  - But it can take a while to balance first.
- What if we are inserting and shard key is increasing?
  - All writes go to same shard { sk: x } ->{ sk: maxKey }
  - 50% of chunks are then moved as every other chunk gets copied to shard 2
  - There needs to be 3X number of writes!
  - You could shard by the md5 hash of the shard key instead for random distribution.

ObjectIDs start with a timestamp and are the default _id value in MongoDB. Each will mostly be just a little bigger than a previous one.
So if you shard by the default _id each write will go to the same chunk as the previous one, and the balancer will be moving stuff
to try and keep the chunks balanced. For two shards 50% of document will be written (To one shard) then read , rewritten to the second shard and deleted from the first.
It is important to not have incrementing shard keys likes this but to ensure they have a good distribution over your shards.
There are a number of techniques to achieve this but it's a common mistake.



The story of Alex

Case study example



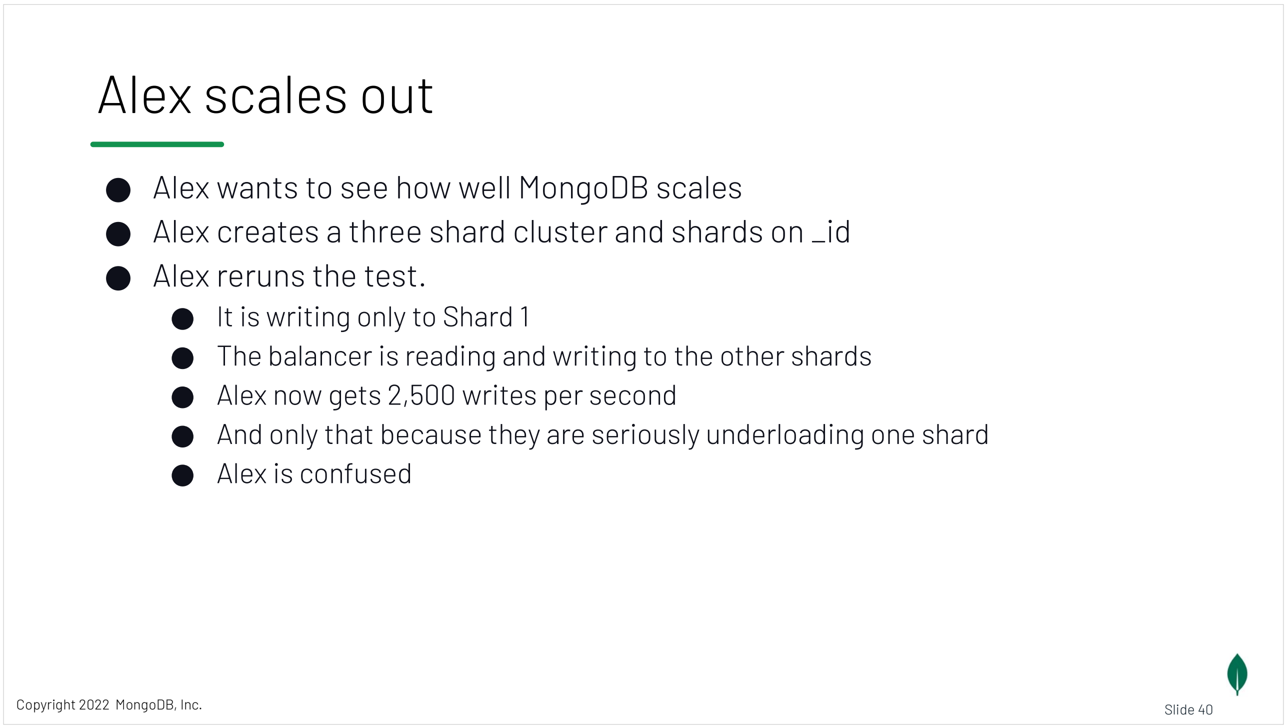
How Alex Learns MongoDB

- Alex wants to see if MongoDB is Fast
  - Alex has heard it's very fast
  - Alex writes something in the shell to test

```
while(true){
    db.test.insertOne({txt:"Is this fast?"})
}
```

Alex is impressed by 3,000 inserts per second.

Case study example



Alex scales out

- Alex wants to see how well MongoDB scales
- Alex creates a three shard cluster and shards on _id
- Alex reruns the test.
  - It is writing only to Shard 1
  - The balancer is reading and writing to the other shards
  - Alex now gets 2,500 writes per second
  - And only that because they are seriously underloading one shard
  - Alex is confused

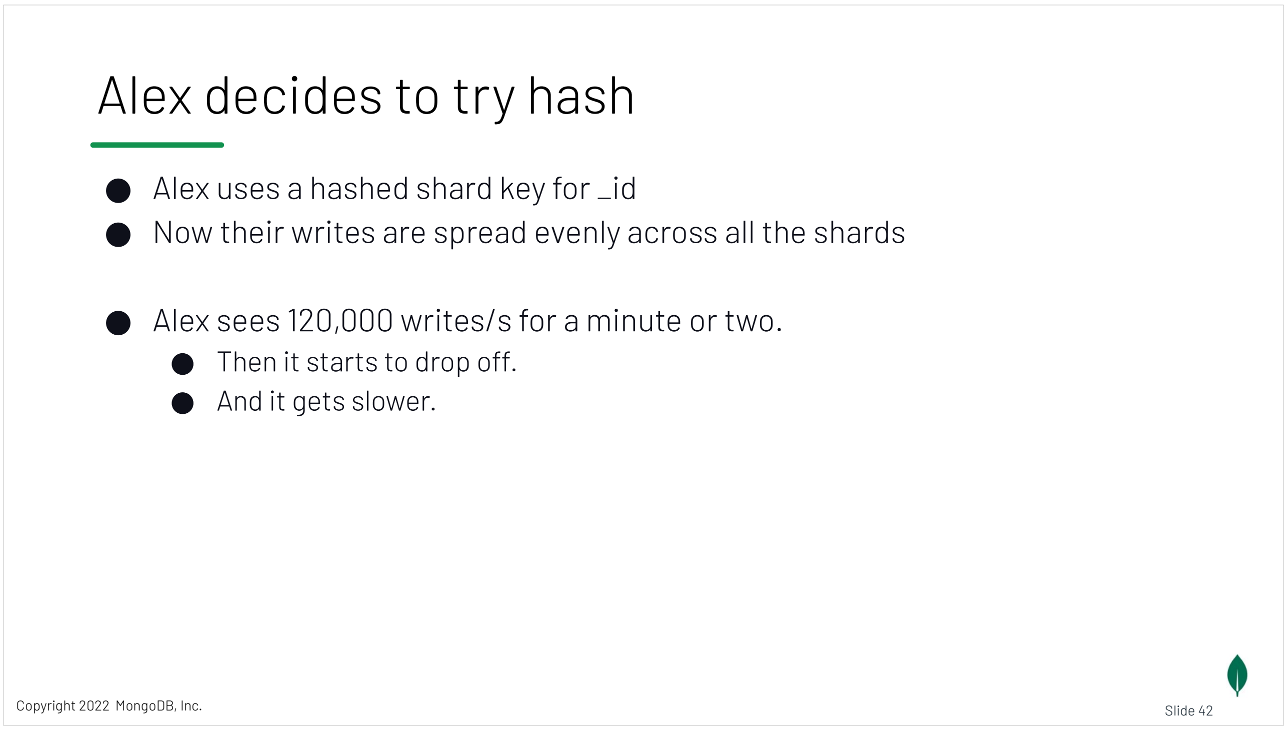
Case study example

Alex tries YCSB

- Someone explains to Alex that they need:
  - A multi-threaded load generator
  - Something faster than Javascript
  - To load data in batches not one document at a time.
- So Alex tries again with YCSB a well known NoSQL Benchmark tool.

- This time a single Replica set manages 40,000 per second
  - Alex is *amazed*! 3,000 seemed good.
- However, the three shards manage just 20,000 between them!
  - Alex goes away and checks stack overflow.
  - Alex sees this happens if all writes go to one chunk due to an increasing shard key. Stack overflow recommends using a hashed shard key.

YCSB - the Yahoo Cloud Serving Benchmark is actually a benchmark for Key/Value capability. Written in 2010 it was never a good way to measure performance of a document store as it misses a lot of the capabilities and is in no way optimised for the model.
It's frequency used to compare different technologies to each other where no valid comparison can be made.



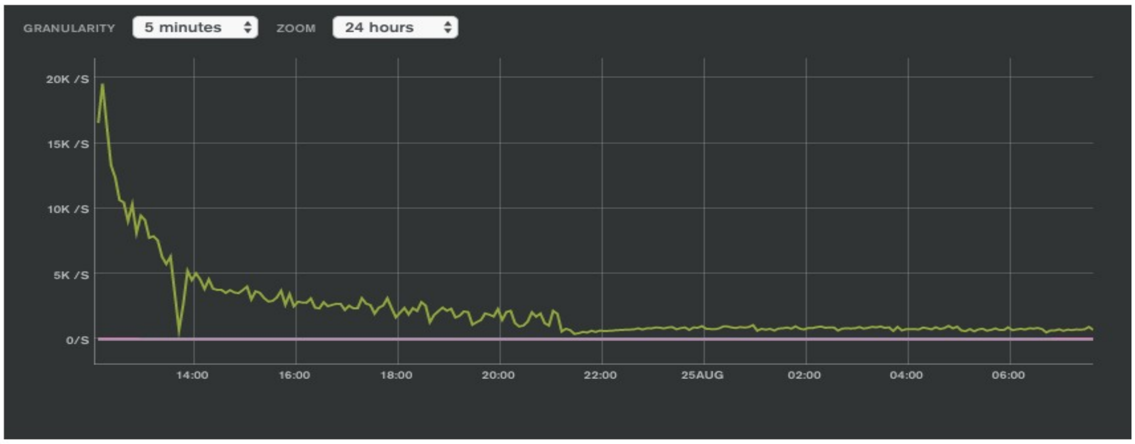
Alex decides to try hash

- Alex uses a hashed shard key for _id
- Now their writes are spread evenly across all the shards

- Alex sees 120,000 writes/s for a minute or two.
  - Then it starts to drop off.
  - And it gets slower.

Case study example

The impact of hashed sharding



- This is from a customer - who asked for help because their data loading did not keep a constant speed even over a few hours.
- Between 12:00 and 21:15, the rate dropped as the amount of disk i/o increased.
- At approx 21:15, the index size grew larger than RAM, at which point it got very slow. This is impacted by disk speed - faster SSDs aren't as bad.

Downsides of hashed indexes

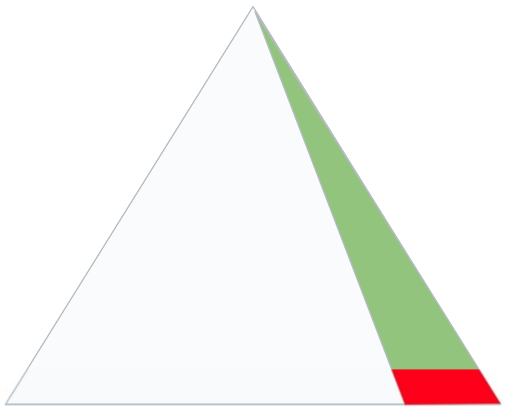
The index is a BTree with internal and leaf nodes.

White - data can be on disk.

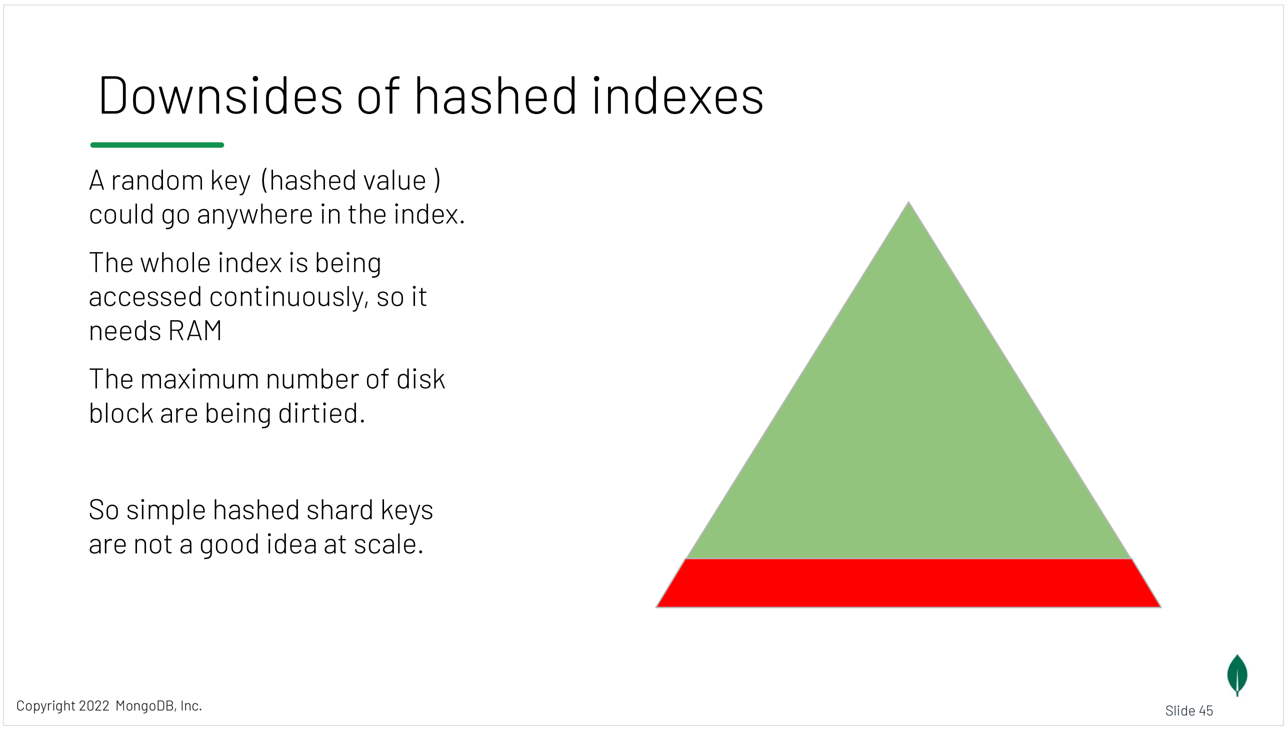
Green - frequently read, some writes, should be in RAM.

Red - frequently written, needs Disk IO.

With an unhashed index an Increasing key goes to the right of the tree.



Case study example



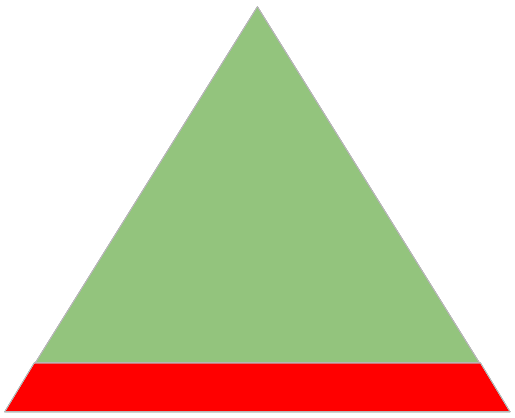
Downsides of hashed indexes

A random key (hashed value) could go anywhere in the index.

The whole index is being accessed continuously, so it needs RAM

The maximum number of disk block are being dirtied.

So simple hashed shard keys are not a good idea at scale.

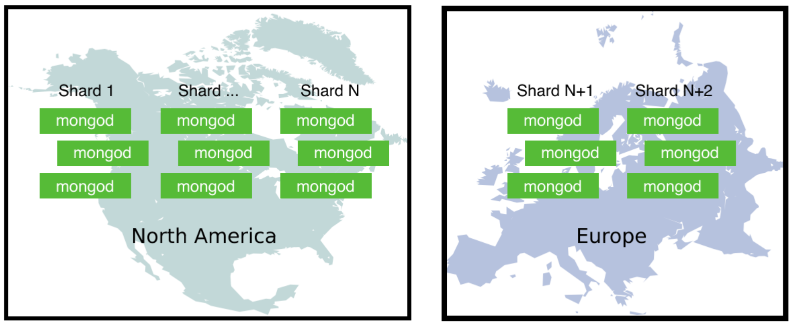


This also applies to other random values like GUIDs in indexes

Common sharding challenges

- Our data may be badly distributed
  - The cluster will eventually balance volume as long as there is a reasonable cardinality
  - However we can get 'Hot' shards doing a lot work relative to others.
  - Hashed sharding avoids these hot shards but takes more resources overall.
  - Our obvious shard key may be a random value anyway (e.g. An MS GUID) and therefore take too many resources.
- Scaling out a cluster can temporarily take a lot of resources
  - Just when we are running out of resources and need to scale out!
  - It's simple to add another shard or two, but those are initially empty.
  - The balancer is responsible for copying data over to them.
  - It is far from instant - it can take days or weeks as lots network and I/O !

Zone based Sharding



Zone-based Sharding provides Instruction to the balancer where to store specific ranges of data. It is a way of controlling what the balancer does.

It is used for two things:

1. Providing better hardware for 'more important' data
2. Keeping data on specific servers for example in specific countries (Regulatory requirements)

Remember, you cannot write to a secondary. So how do you keep US data in the US and European data in europe but it appears as a single collection.
You have multiple shards, sharded by location. One for the USA one for Europe for example. Each is hosted in their own region.
The shard key determines if data should be stored in the US or Europe. You might have a 'Secondary' in the other region to make querying quicker.
.
Supported through the Atlas GUI for ease of setup (Screenshot)

Sharding for Parallelism

Unusual use case but powerful when used.

- Running Aggregations and some queries - reads sent to all shards
- Bad for OLTP / Normal queries
- But for a small number of analytics queries - the more CPUs the better
- Sometimes we run many shards on one server! (microsharding). This works where:
  (a) All data can fit in RAM and therefore disks not a bottleneck
  (b) You have a small number of power users - more CPUs than users
  (c) You know how to write aggregations that can use parallelism.

Clarification on (b) more CPUs than users:

- The query engine doesn't parallelize a single query, so if a query is perform on a non-sharded collection, it will effectively use one CPU core out of potentially many.
- If a collection is queried at a high rate, using microsharding effectively allows users to use more than one core for one query, because the query is split between multiple shards (assuming it's not targeted to a single shard in the first place).

Recap

- Sharding is a mechanism of partitioning large collections across multiple servers.
- It is important to choose the shard key well.
- Sharding is typically used in a system more than 2TB in size
- Planning for Sharding early can help a lot when you need it.
- Many people struggle with Sharding due to not understanding what you do now.
- You can also shard for geographic reasons.