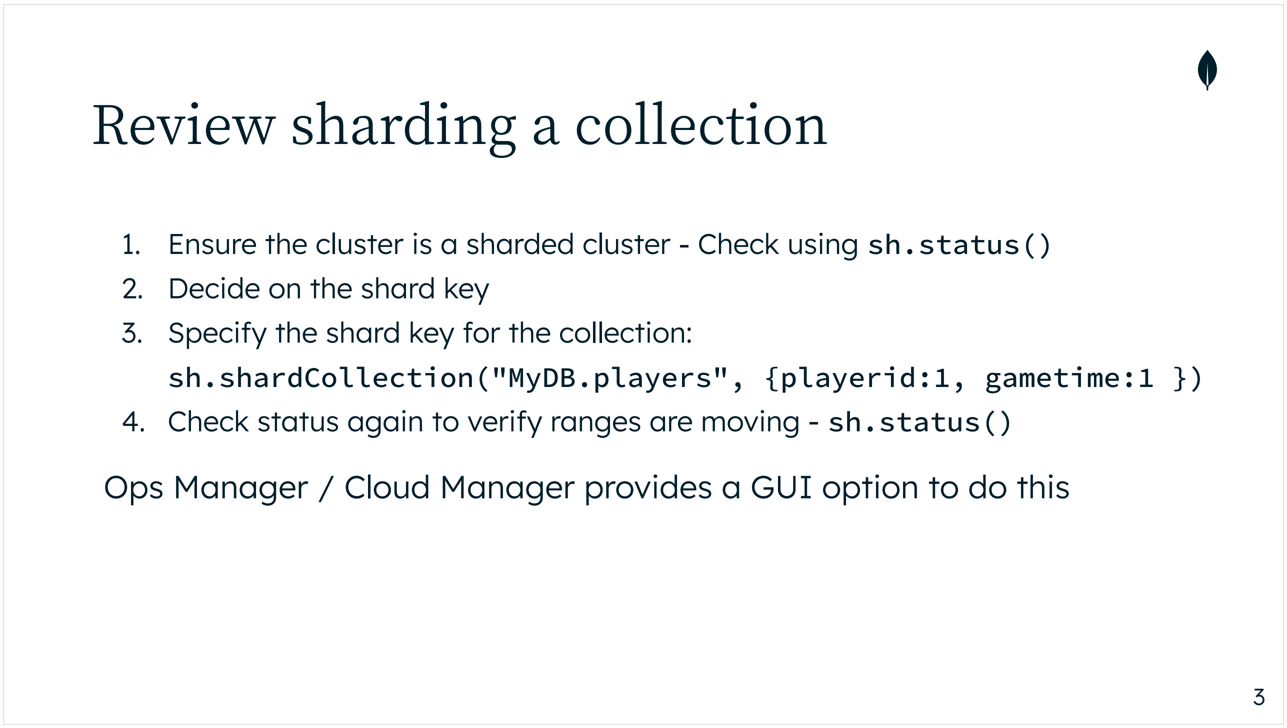
MDB300

Sharding Part II

MongoDB Production Readiness

Release: 20250117



Review sharding a collection

1. Ensure the cluster is a sharded cluster - Check using `sh.status()`
2. Decide on the shard key
3. Specify the shard key for the collection:
   `sh.shardCollection("MyDB.players", {playerid:1, gametime:1 })`
4. Check status again to verify ranges are moving - `sh.status()`

Ops Manager / Cloud Manager provides a GUI option to do this

On MongoDB version lesser than 6.0 enable sharding on the database using the following command before Step 3.

sh.enableSharding("MyGameDB")

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

How config metadata caching works

- Driver sends operation to **mongos**
- **mongos** has a cached copy of config data which is versioned and 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 RS's knows

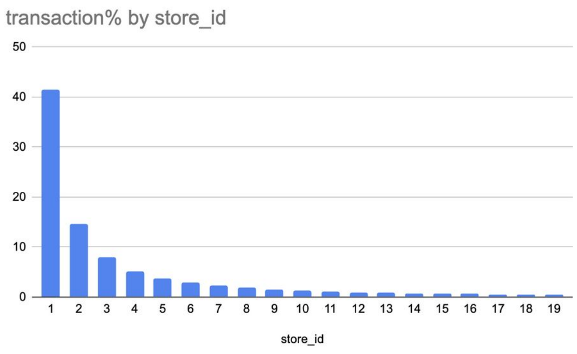
- Replica Set ignores operation and tells mongos to refresh the cache
- mongos does so, then re-sends operation to the correct servers

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.

Balancing data

Data and query imbalance

- What if some stores do a lot more business than others?
- Can we still assign key range 1-33 to a single shard?
- What happens when there is data imbalance?
  - Degradation in cluster throughput
  - Longer backup times
- But real world data always has imbalance



- The shard key can be changed for a collection (reshard) to change the distribution of your data across a cluster (From MDB v5+)

Sharded collection data is stored in key ranges.

For example, Customer number 667726 might be in a key range going from 660000 to 670000, and that range is stored on shard 8. Inserts, updates, or queries for that value will be sent just to shard 8.
From MongoDB 8.0, you can reshard a collection on the same shard key, allowing you to redistribute data to include new shards or to different zones without changing your shard key.

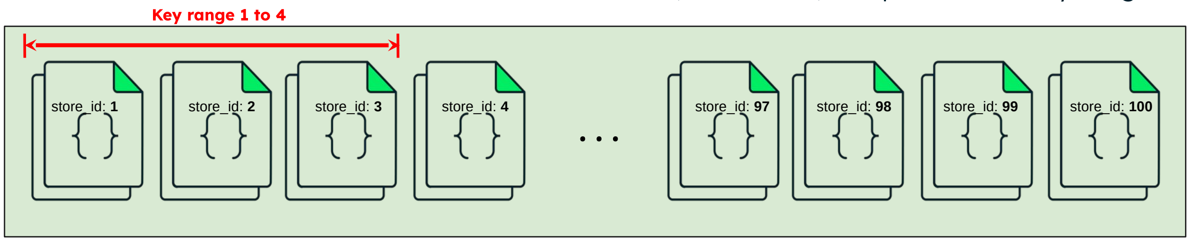
Automatic balancing in MongoDB

- MongoDB migrates data ranges in a sharded cluster to distribute the data of a sharded collection evenly among shards
  - Manual
  - Automatic
- The balancer (background process) balances data across the cluster
- Migrates key ranges if amount of data between the largest and smallest shard exceed a certain threshold

The balancer runs on the primary node of the config server RS.

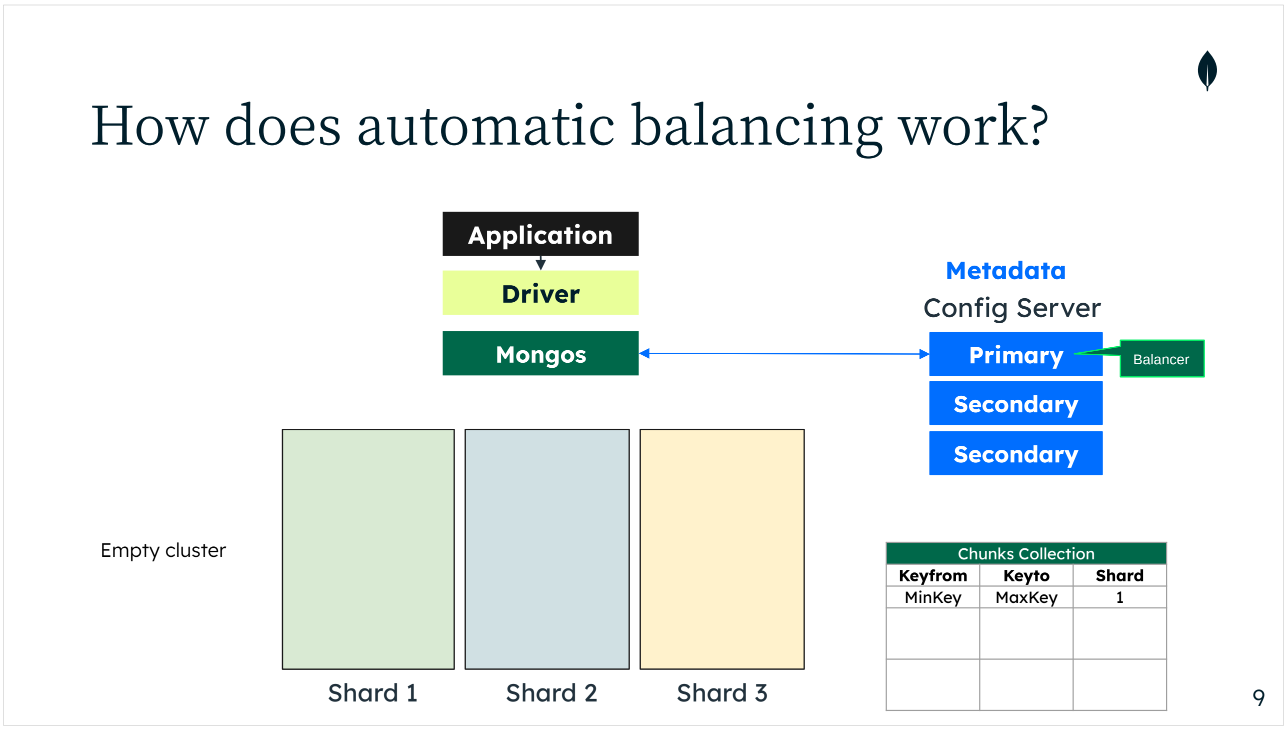
Key ranges

- Key range is the unit of balancing in MongoDB
- When we define a portion of data for migrating between shards, we define it in terms of key ranges (Default 128MB)
- For example, store_id 1 to 4 is a key range:
  - 1 is the lower bound and is included
  - 4 is the upper bound and is excluded
- Documents with store_ids between 1 and 4 (4 excluded) are part of this key range



The amount of data moved by the balancer at a time is a configurable parameter. The default size is 128MB. The balancer will only move key ranges that are equal or smaller than this default size.

When key ranges become larger than this value and are indivisible due to containing only a single key value (jumbo key range) then the balancer will not be able to move such a key range.

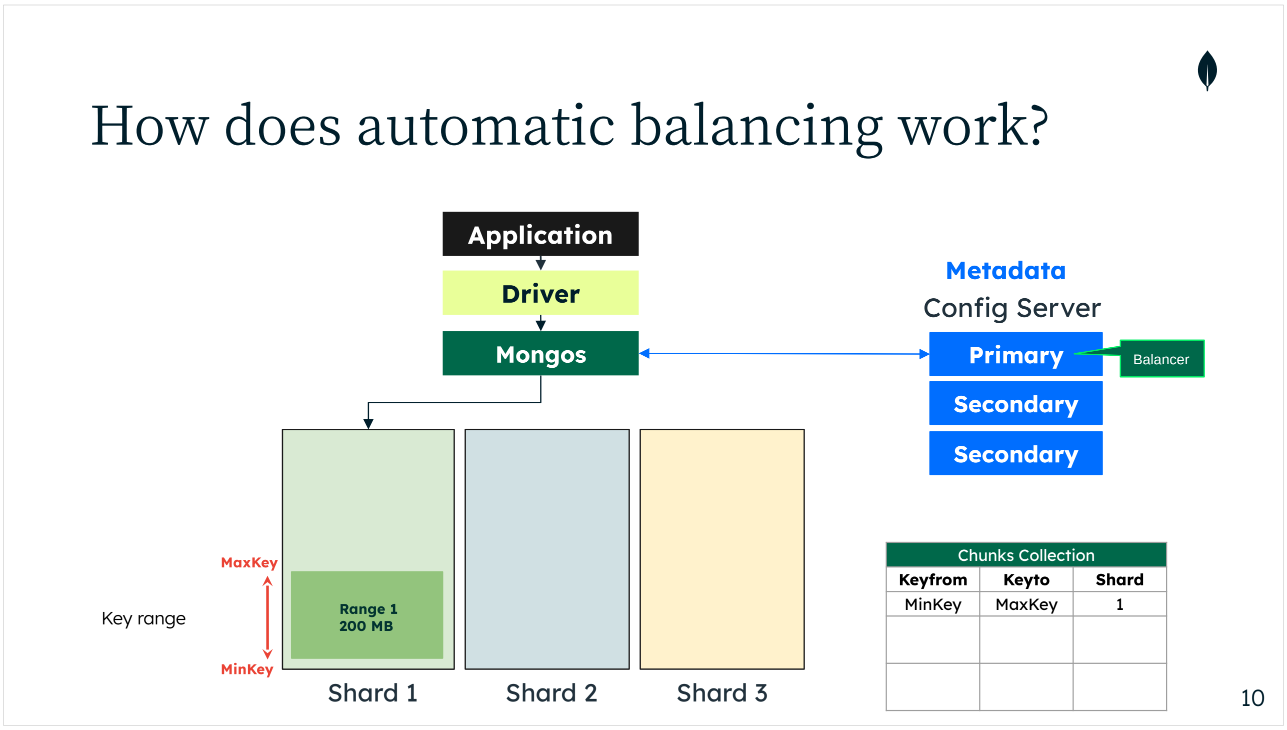


How does automatic balancing work?

Application
Driver
Mongos
Metadata
Config Server
Primary
Balancer
Secondary
Secondary
Empty cluster
Shard 1
Shard 2
Shard 3

Chunks Collection		
Keyfrom	Keyto	Shard
MinKey	MaxKey	1

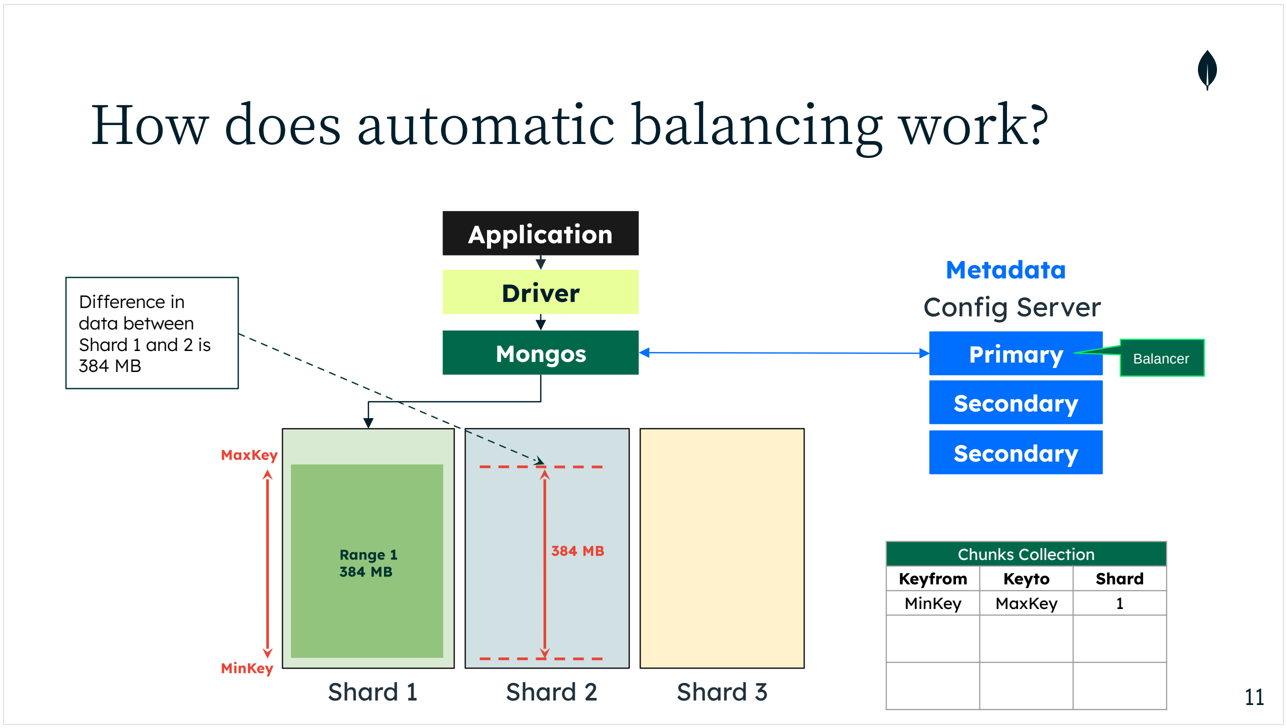
Let us take a look at how automatic balancing work in MongoDB.
Let us start with an empty 3-sharded cluster.



How does automatic balancing work?

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

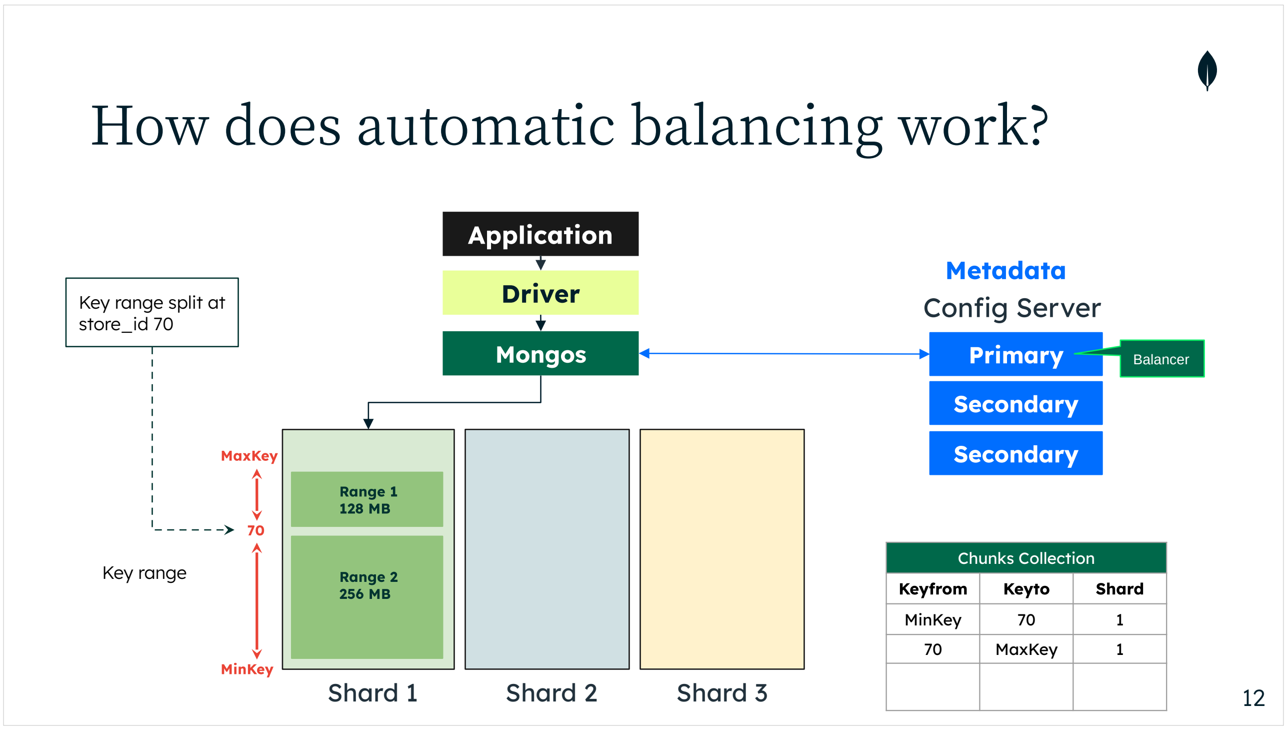
With a newly sharded collection, it contains just one key range from MinKey to MaxKey, which will hold all documents.
This key range will exist only on one shard because key ranges cannot be duplicate or overlapping. Therefore all writes initially go to a single shard (Shard 1).



How does automatic balancing work?

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

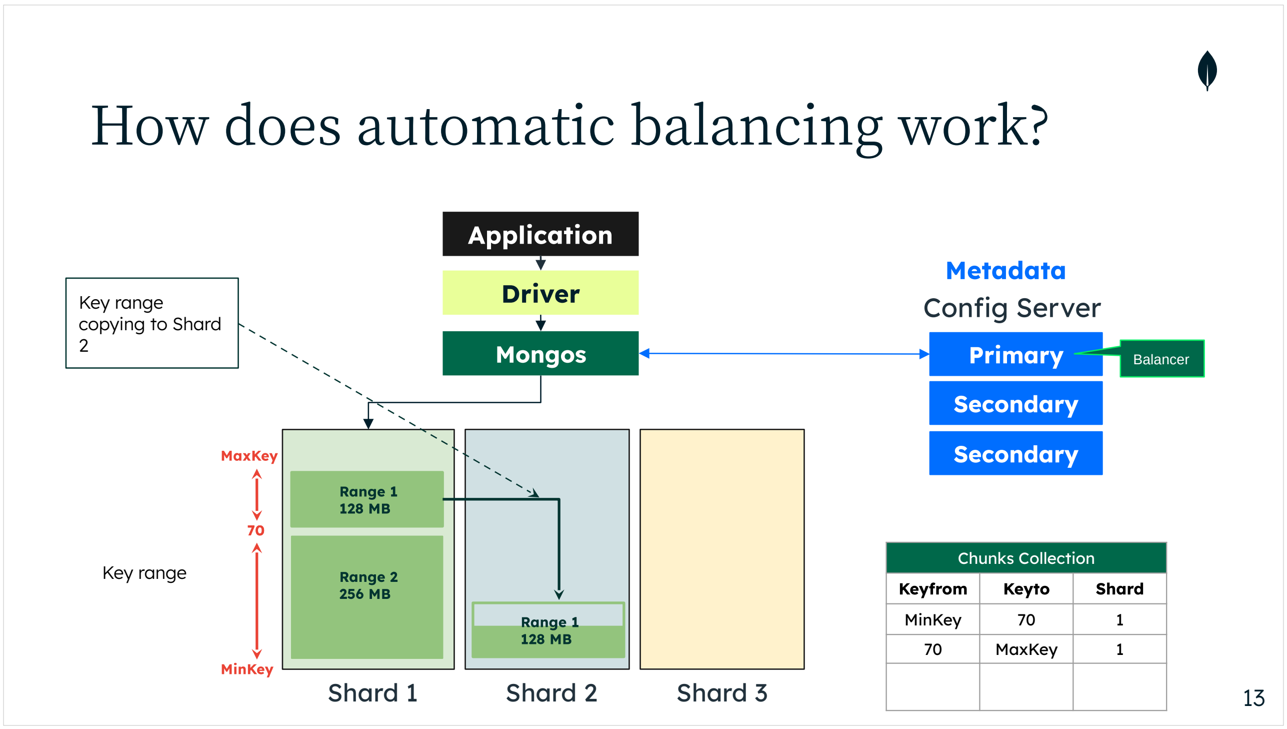
As the data grows and when the difference in data between the smallest and the largest shard becomes 3 times the max size of default key range (in this case 384MB) the balancer kicks in.



How does automatic balancing work?

Chunks Collection		
Keyfrom	Keyto	Shard
MinKey	70	1
70	MaxKey	1

The balancer finds a split point within the key range in such a way that the size of the key range to me moved is smaller than the default key range (default 128MB).

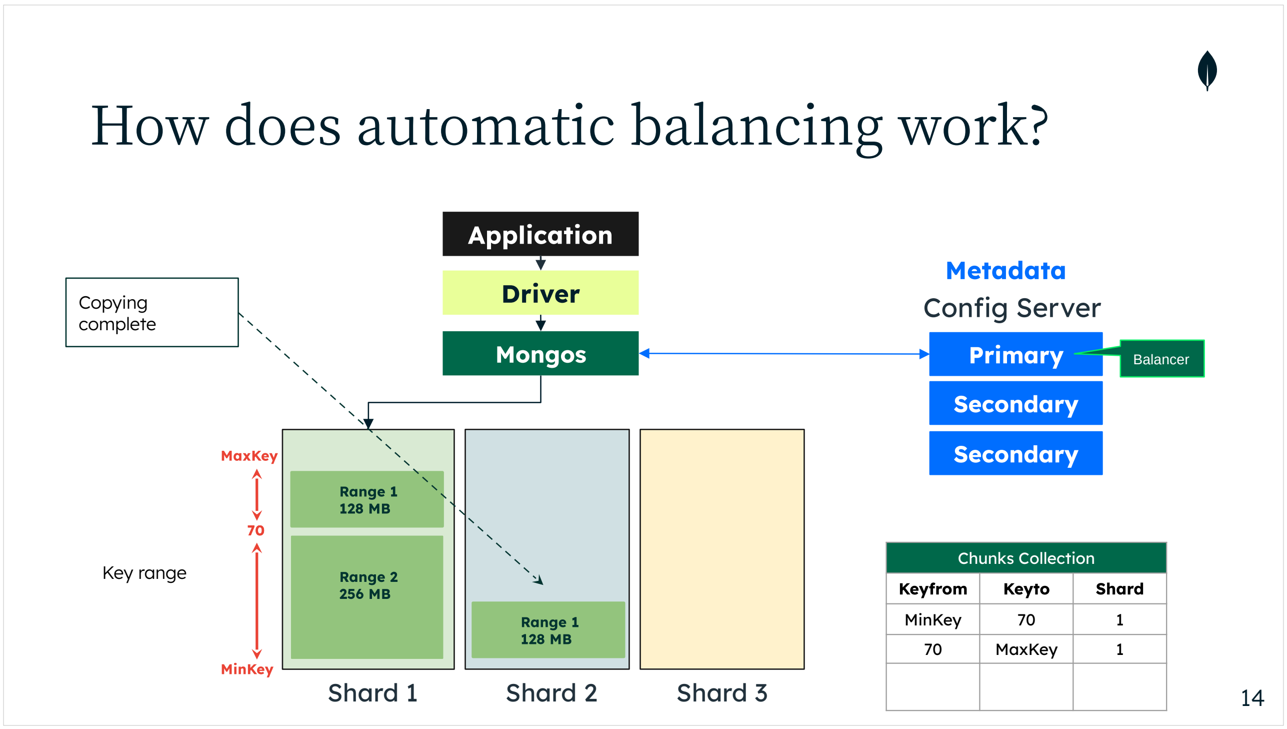


How does automatic balancing work?

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	70	1
70	MaxKey	1

The key range is copied to the shard with the least amount of data.
While the key range is being migrated the original key range will be still in use in shard 1.
Any changes to the key range during copying will be done on the source shard.
After the initial copying is complete the deltas are copied if anything was changes in the source range during the copy process.

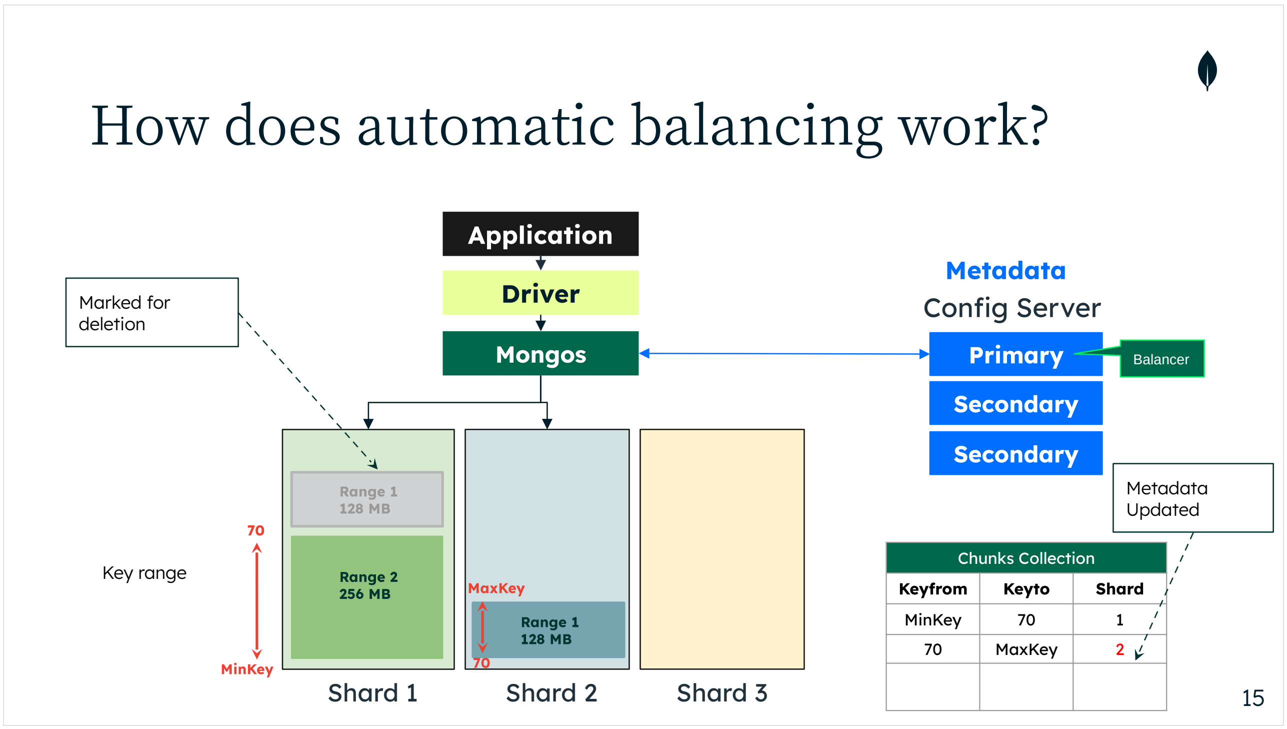


How does automatic balancing work?

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	70	1
70	MaxKey	1

Once all the data from the key range has been copied to the destination shard, then the new shard (Shard 2) is ready to accept the queries for the key range.



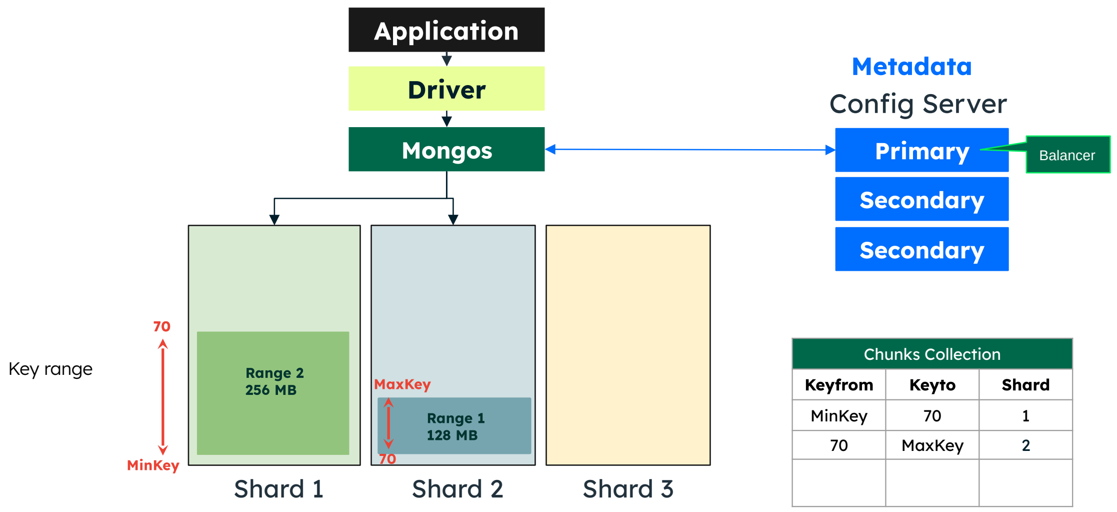
How does automatic balancing work?

Chunks Collection

Keyfrom	Keyto	Shard
MinKey	70	1
70	MaxKey	2

The metadata is updated to show that keyrange 70 to Max key is now located on shard 2.
Queries accessing the key range can now be routed to shard 2.
The source key range is marked for deletion.

How does automatic balancing work?



Chunks Collection

Keyfrom	Keyto	Shard
MinKey	70	1
70	MaxKey	2

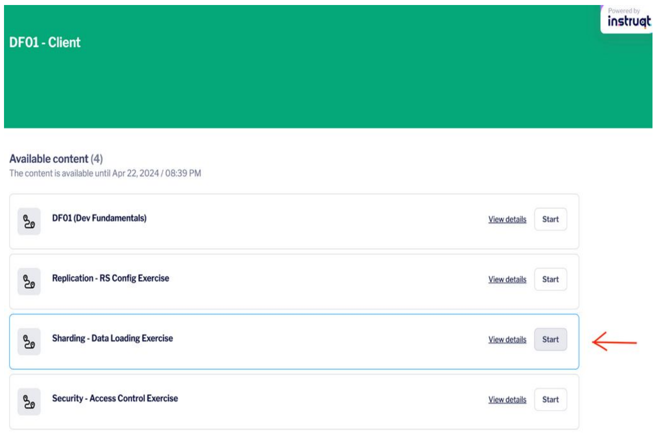
Exercise - Data Loading

In this lab, you will be working with a 2 shard cluster configured with a mongos and a config server.

You will import a larger data set and shard a collection.

You will run queries that will allow you to see which shards are being accessed to retrieve the results.

Start up the lab titled Sharding - Data Loading on Instruqt


DF01 - Client

Available content (4)
The content is available until Apr 22, 2024 / 08:39 PM

DF01 (Dev Fundamentals)
Replication - RS Config Exercise
Sharding - Data Loading Exercise
Security - Access Control Exercise

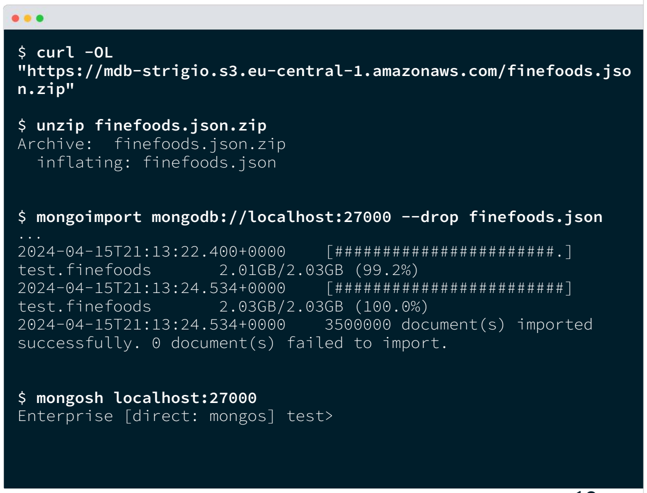
NOTE! - this activity is only intended for use in instruqt! Do not use this with strigo. If the instructor has not configured the training as a multi-track live event, then simply start an additional event with the lab title sharding - data loading exercise.

Exercise - Data Loading

Load the sample dataset & unzip it

Import the dataset into the mongos

Login to the **mongos** server using the mongosh shell



```
$ curl -OL
"https://mdb-strigio.s3.eu-central-1.amazonaws.com/finefoods.json.zip"

$ unzip finefoods.json.zip
Archive:  finefoods.json.zip
  inflating: finefoods.json


$ mongoimport mongodb://localhost:27000 --drop finefoods.json
...
2024-04-15T21:13:22.400+0000    [#######################.]
test.finefoods       2.01GB/2.03GB (99.2%)
2024-04-15T21:13:24.534+0000    [########################]
test.finefoods       2.03GB/2.03GB (100.0%)
2024-04-15T21:13:24.534+0000    3500000 document(s) imported
successfully. 0 document(s) failed to import.


$ mongosh localhost:27000
Enterprise [direct: mongos] test>
```

Learners have been provisioned with a 2 shard cluster (no replication). In this activity, we will be loading a large data set into the sharded cluster and will take a look at how the data is balanced by the automatic load balancer once a shard key has been chosen

Configuration is
mongos: 27000
config server: 27001
shard_a: 27002
shard_b: 27003

This dataset contains 3500000 documents so the mongoimport can take a few minutes to finish.

Exercise - Data Loading

Verify your sharded cluster using **`sh.status()`**

Before version 7.0, use **`sh.enableSharding("coll")`** to enable sharding for the database that contains the collection to be partitioned.

```
[mongos]test > sh.status()
...
[{ _id: 'shard_A',
    host: 'shard_A/localhost:27002',
    state: 1,
    topologyTime: Timestamp({ t: 1713216972, i: 2 })
  },
  {
    _id: 'shard_B',
    host: 'shard_B/localhost:27003',
    state: 1,
    topologyTime: Timestamp({ t: 1713216974, i: 2 })
  }]
...


> sh.enableSharding("test")
{ok: 1}, ...
```

sh.status() will show the status of the sharded cluster similar to the rs.status command for a replica set. Here you will see the configuration of the hosts and what their role is within the cluster. The 2 shards are empty and have no existing datasets on them.
sh.enableSharding(database, [optional - primaryShard]) explicitly creates a database. This is not required from v6+

Exercise - Examine the schema

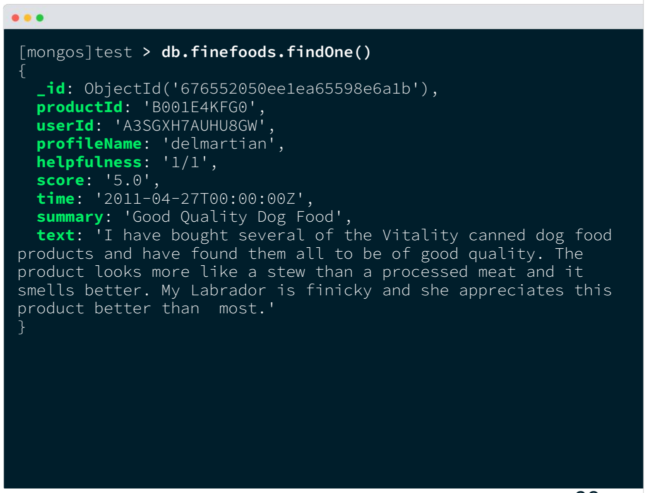
Examine the document schema to create an index

Assume the most common queries use productId and userId

Which fields are best suited as the shard key?

Remember to select a key with

- High cardinality,
- Non-monotonically increasing
- Low Frequency



```
[mongos]test > db.finefoods.findOne()
{
  _id: ObjectId('676552050ee1ea65598e6a1b'),
  productId: 'B001E4KFG0',
  userId: 'A3SGXH7AUHU8GW',
  profileName: 'delmartian',
  helpfulness: '1/1',
  score: '5.0',
  time: '2011-04-27T00:00:00Z',
  summary: 'Good Quality Dog Food',
  text: 'I have bought several of the Vitality canned dog food
products and have found them all to be of good quality. The
product looks more like a stew than a processed meat and it
smells better. My Labrador is finicky and she appreciates this
product better than  most.'
}
```

Example document:
```
{
 _id: ObjectId("661da375b0325a984fedca86"),
 productId: 'B001P76YCK',
 userId: 'A3HO7GON4GQNJ7',
 profileName: 'Ms. Mary J. Caudell "Txn Lady"',
 helpfulness: '2/2',
 score: '4.0',
 time: '2011-12-25T16:32:30Z',
 summary: 'Good Quality Dog Food',
 text: 'I have bought several of the Vitality canned dog food products and have found them all to be
of good quality. The product looks more like a stew than a processed meat and it smells better. My
Labrador is finicky and she appreciates this product better than  most.'
}
```
Choosing the right key:

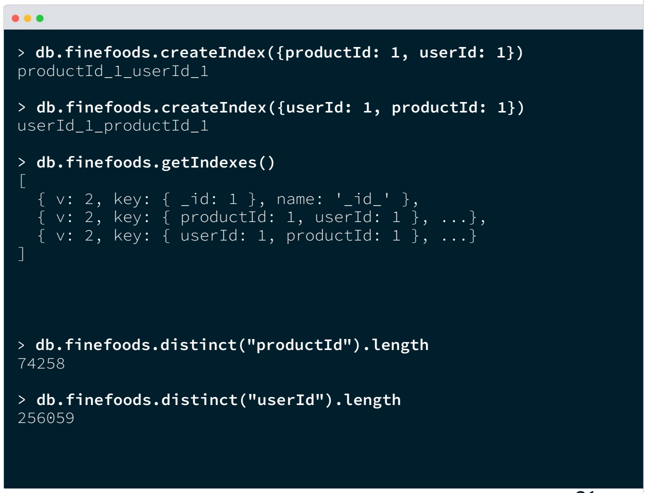
- High Cardinality: An example of a shard key with high cardinality could be the userId or productId. The cardinality of a shard key determines the maximum number of chunks the balancer can create. However, high cardinality does not guarantee an even data distribution across a sharded cluster.
- Non-monotonically Increasing: A shard key that increases monotonically will result in new inserts being routed to the chunk with maxKey as the upper bound (ie., time or _id)
- Frequency: The lower, the better. If many documents share the same shard key, chunks may grow and become indivisible(aka jumbo chunks). Mitigation is the use of a shard key combination that includes low-frequency values. We could use a combined key: productID, userId

Exercise - Creating Indexes

Create the indexes we want to analyze to use one as shard key

- `{productId,userId}`
- `{userId,productId}`

Review the cardinality of each field



```
> db.finefoods.createIndex({productId: 1, userId: 1})
productId_1_userId_1

> db.finefoods.createIndex({userId: 1, productId: 1})
userId_1_productId_1

> db.finefoods.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { productId: 1, userId: 1 }, ...},
  { v: 2, key: { userId: 1, productId: 1 }, ...}
]



> db.finefoods.distinct("productId").length
74258

> db.finefoods.distinct("userId").length
256059
```

Find products by productId and userId with 2 possible indexes to support the queries.
Only one of these indexes is required, but we vary the index field order to learn more about our design decision.
userId has a higher cardinality than the productId field. Given that products can be reviewed by many users, it would be more efficient to filter by user as it is more selective.
The _id field has the highest cardinality but increases monotonically, making it a poor candidate for a shard key.

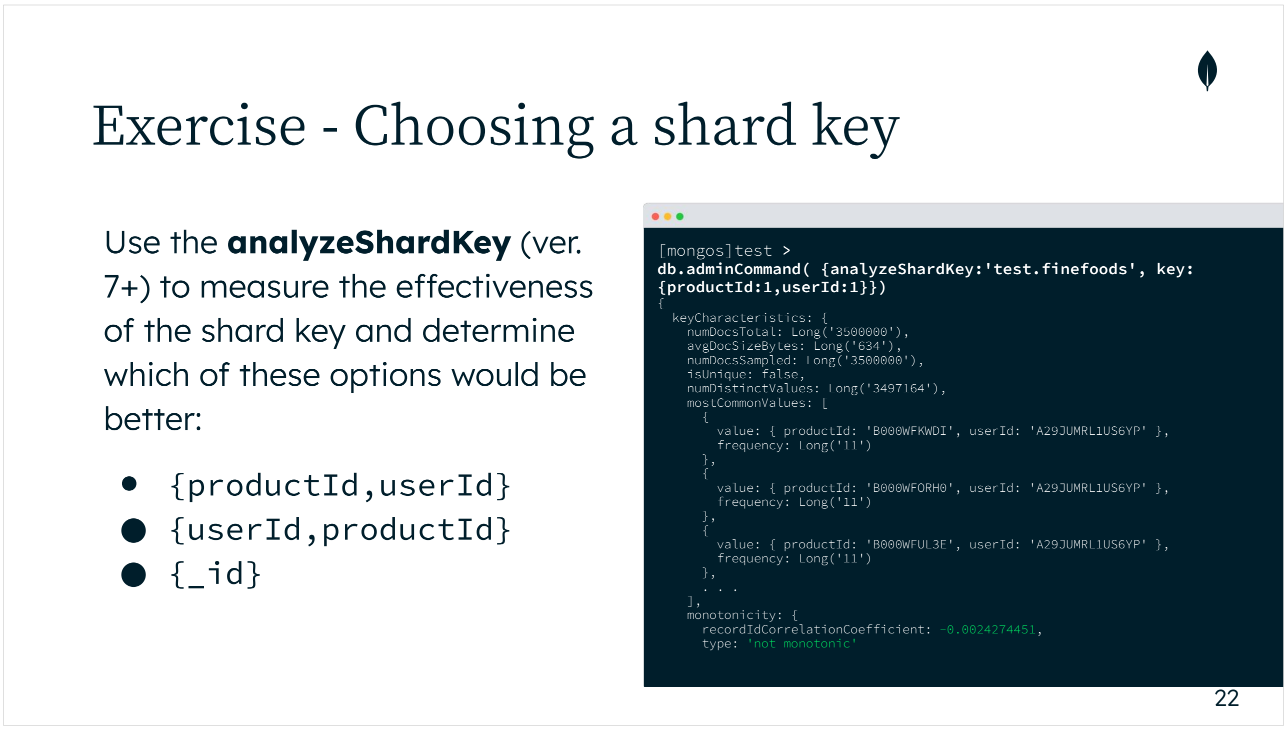
Note, we can also analyze the data to understand how many users and products there are in this dataset to see the cardinality of each field using the distinct() function

Note, if the cardinality is too high distinct will return the following error:distintc too big, 16mb cap

```
db.finefoods.distinct("_id").length
MongoServerError[Location17217]: Executor error during distinct command ::
caused by :: distinct too big, 16mb cap
```

Alternatively we can use the aggregation pipeline as in the example below:

```
db.finefoods.aggregate([
{$group:{_id:null, productId:{$addToSet:'$productId'}, userId:
{$addToSet:'$userId'}}},
{$set:{uniqueProducts:{$size:'$productId'},uniqueUsers:
{$size:'$userId'}}},
{$project:{_id:0, uniqueProducts:1,uniqueUsers:1} }])
```



Exercise - Choosing a shard key

Use the **analyzeShardKey** (ver. 7+) to measure the effectiveness of the shard key and determine which of these options would be better:

- `{productId,userId}`
- `{userId,productId}`
- `{_id}`

```
[mongos]test >
db.adminCommand( {analyzeShardKey:'test.finefoods', key:
{productId:1,userId:1}})
{
  keyCharacteristics: {
    numDocsTotal: Long('3500000'),
    avgDocSizeBytes: Long('634'),
    numDocsSampled: Long('3500000'),
    isUnique: false,
    numDistinctValues: Long('3497164'),
    mostCommonValues: [
      {
        value: { productId: 'B000WFKWDI', userId: 'A29JUMRL1US6YP' },
        frequency: Long('11')
      },
      {
        value: { productId: 'B000WFORH0', userId: 'A29JUMRL1US6YP' },
        frequency: Long('11')
      },
      {
        value: { productId: 'B000WFUL3E', userId: 'A29JUMRL1US6YP' },
        frequency: Long('11')
      },
      . . .
    ],
    monotonicity: {
      recordIdCorrelationCoefficient: -0.0024274451,
      type: 'not monotonic'
```

22

analyzeShardKey returns different metrics depending on the keyCharacteristic and readWriteDistribution values you specify when you run the method.

The `recordIdCorrelationCoefficient` helps determine how well the chosen shard key will distribute data across shards using the correlation between shard key values and the record IDs of documents. This can range from -1 to 1. 1 indicates a shard key is likely to lead to an uneven distribution as it is monotonic. A value closer to 0 means the there will be a good distribution of data across the shards. A value of -1 is rare and not likely to occur as it would mean the data distribution is reversed in some way.

Run different tests:
//1st shard key candidate
```
db.adminCommand({analyzeShardKey:'test.finefoods',key:
{productId:1,userId:1}})
```

Output:
```
   recordIdCorrelationCoefficient: -0.0024274433,
   type: 'not monotonic'
```

//2nd shard key candidate
```
db.adminCommand({analyzeShardKey:'test.finefoods',key:
{userId:1,productId:1}})
```

Output:
```
   recordIdCorrelationCoefficient: -0.000401584,
   type: 'not monotonic'
```

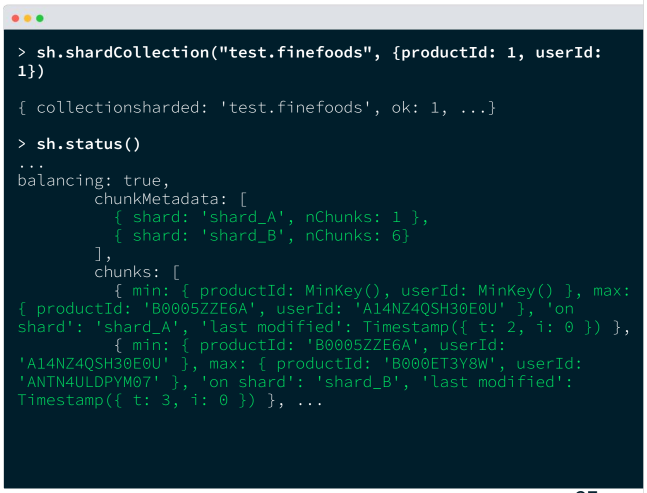
//3rd the control candidate
```
db.adminCommand({analyzeShardKey:'test.finefoods',key:{_id: 1}})
```

Output:

Exercise - Sharding the collection

Shard the collection and monitor how the load balancer distributes the data across the shards

Notice the ranges that are being created and in which shard they are placed in.



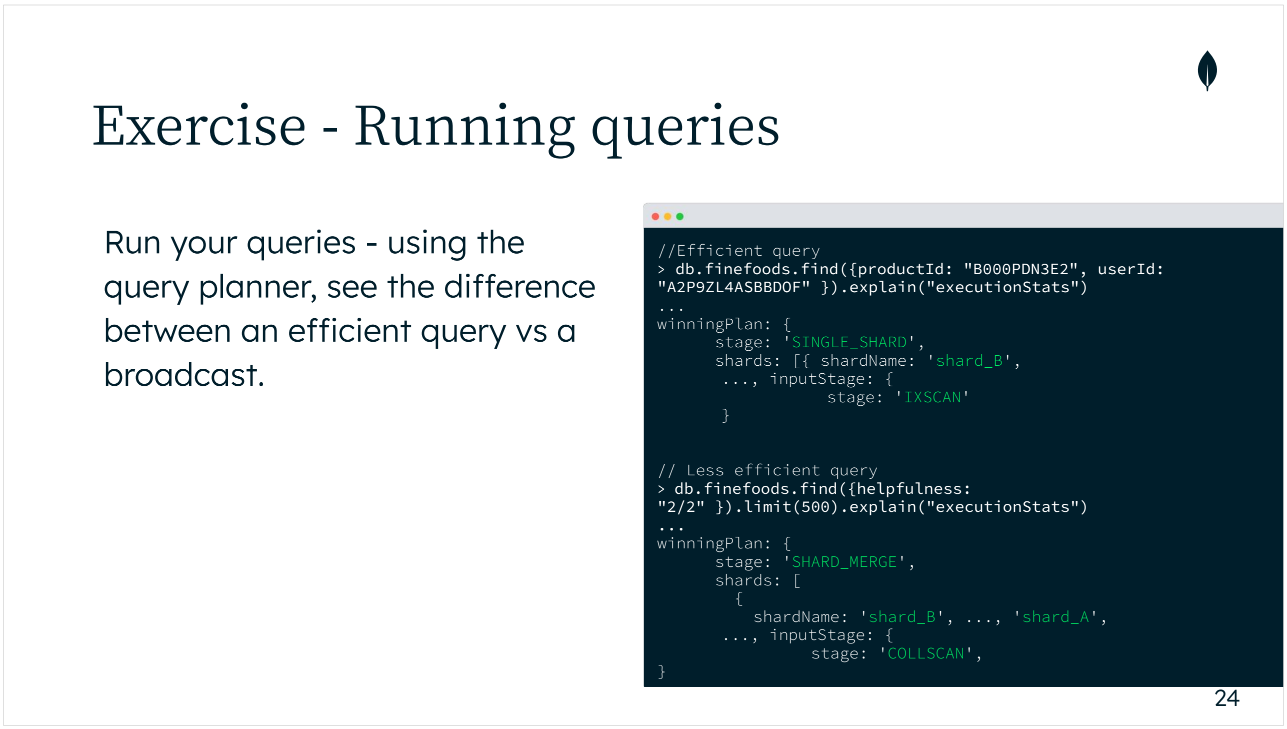
```
> sh.shardCollection("test.finefoods", {productId: 1, userId:
1})

{ collectionsharded: 'test.finefoods', ok: 1, ...}

> sh.status()
...
balancing: true,
        chunkMetadata: [
          { shard: 'shard_A', nChunks: 1 },
          { shard: 'shard_B', nChunks: 6}
        ],
        chunks: [
          { min: { productId: MinKey(), userId: MinKey() }, max:
{ productId: 'B0005ZZE6A', userId: 'A14NZ4QSH30E0U' }, 'on
shard': 'shard_A', 'last modified': Timestamp({ t: 2, i: 0 }) },
          { min: { productId: 'B0005ZZE6A', userId:
'A14NZ4QSH30E0U' }, max: { productId: 'B000ET3Y8W', userId:
'ANTN4ULDPYM07' }, 'on shard': 'shard_B', 'last modified':
Timestamp({ t: 3, i: 0 }) }, ...
```

Based on the schema, the index we are using to shard the collection is {productId: 1, userId: 1}

A common pitfall to sharding is that it can take a while depending on how its done and how large the dataset is.
Running sh.status() will reflect the latest status of the sharded cluster



Exercise - Running queries

Run your queries - using the query planner, see the difference between an efficient query vs a broadcast.

```
//Efficient query
> db.finefoods.find({productId: "B000PDN3E2", userId:
"A2P9ZL4ASBBDOF" }).explain("executionStats")
...
winningPlan: {
      stage: 'SINGLE_SHARD',
      shards: [{ shardName: 'shard_B',
       ..., inputStage: {
              stage: 'IXSCAN'
       }


// Less efficient query
> db.finefoods.find({helpfulness:
"2/2" }).limit(500).explain("executionStats")
...
winningPlan: {
      stage: 'SHARD_MERGE',
      shards: [
        {
          shardName: 'shard_B', ..., 'shard_A',
       ..., inputStage: {
             stage: 'COLLSCAN',
}
```

db.finefoods.find({productId: "B000PDN3E2", userId: "A2P9ZL4ASBBDOF" }).explain("executionStats").executionStats - will show only the executionStats object and reduce the noise of the output from the query planner

The less efficient query has a limit applied as the query planner will wait until it is able to gather all the documents that match vs. iteratively going through them via the cursor without the .explain(). Ensure you do this to avoid hitting a potential memory issue on the provided container.

The configuration files can be found in the /etc/ directory if learners are curious about how this sharded cluster was configured. Each host would have its distinct configuration file.

Presplitting before bulk loading

When bulk loading data to an empty sharded collection all the data will be written to the same initial shard and then parts will be migrated to other shards as an imbalance is created.

Presplit the data to avoid the initial shuffling via the first shard

- Manually move key ranges to other shards before loading data
- This can speed up a bulk load of data

One thing we can do is, before loading any data move key ranges to other shards
Doing this generally requires us to have an idea of the range of values in the shard key.
Moving an empty key range takes very little time as there is no data to copy, it's just a metadata change.
This is done with the commands moveRange
We only need to create one range per shard, especially if we move them manually.

Sharding Pitfalls (1 of 2)

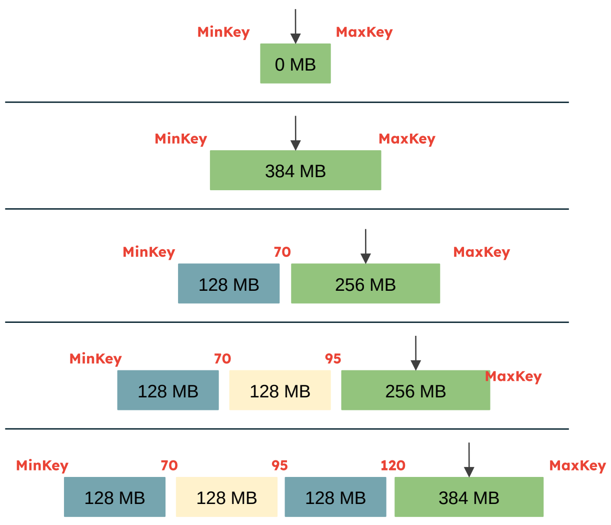
Balance required additional IO

- Each key range that moves writes three times (write on 1, write on 2, delete on 1)
- Getting the deltas takes time in a busy system (more reads and writes)
- Once key ranges are distributed, if writes hit random shards then is good - self balanced
- First balance can take a while

Sharding Pitfalls (2 of 2)

What if we are inserting and shard key is increasing?

- All writes go to same shard `{ sk: x } ->{ sk: maxKey }`
- 50% of key ranges are moved as every other key ranges gets copied to other shard - 3X writes!
- Choose a compound shard key such that it is not monotonically increasing
- Else shard by the MD5 hash of the shard key instead for



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 shard as the previous one, and the balancer will be moving stuff
to try and keep the data 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.

Common sharding challenges

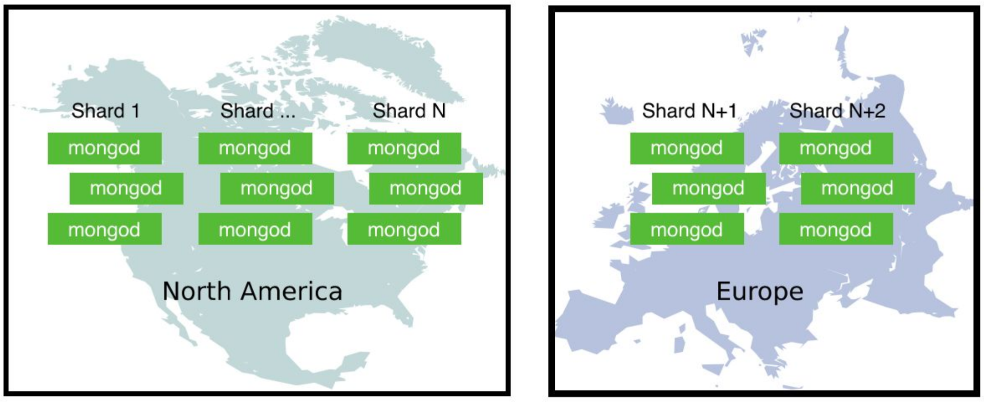
Data may be badly distributed

- The cluster eventually balances volume as long as there is a reasonable cardinality
- Can get 'Hot' shards doing a lot work relative to others
- Hashed sharding avoids hot shards but takes more resources overall
- The shard key may be a random value (e.g. MS GUID) taking too many resources

Scaling out a cluster can temporarily take a lot of resources

- Running out of resources implies a need to scale out!
- Easier to add more shards, but those are initially empty
- The balancer is responsible for copying data over to them
- Can take days or weeks as lots network and I/O !

Zone based sharding

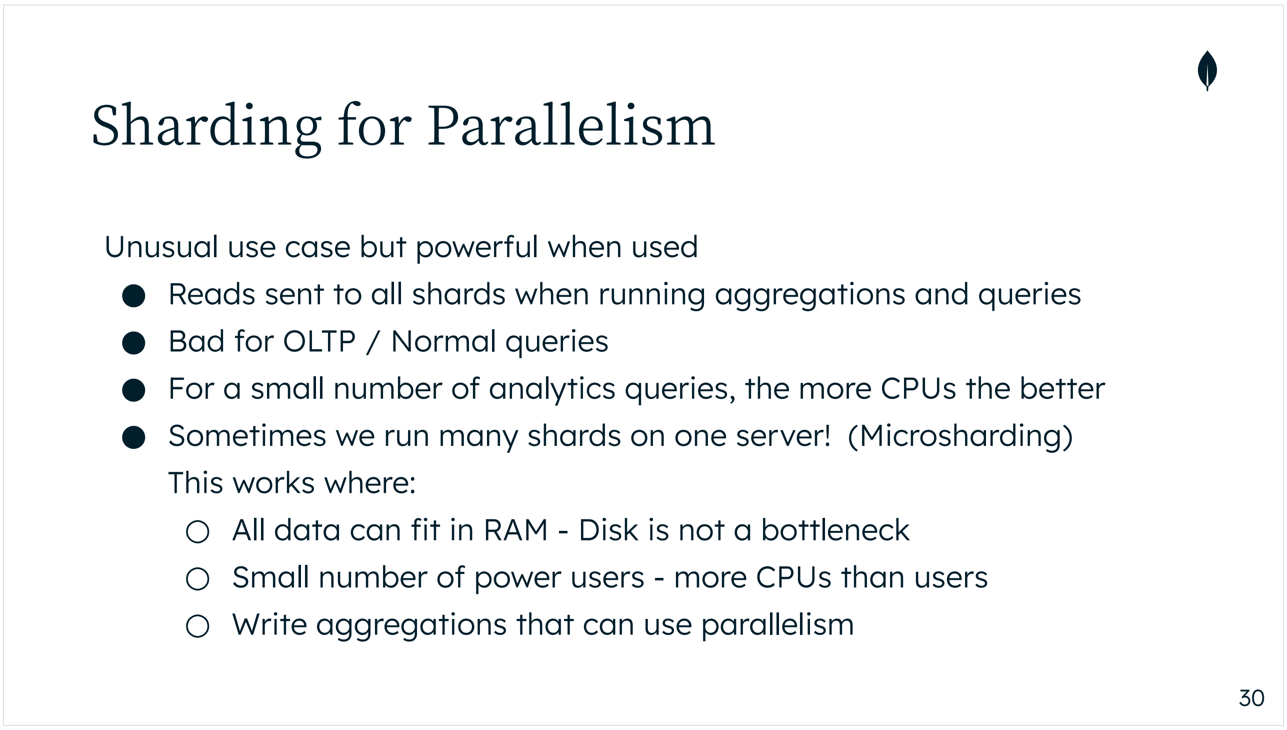


Zone-based Sharding provides Instruction to the balancer where to store specific ranges of data. It is a way of controlling what the balancer does.

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

- Reads sent to all shards when running aggregations and queries
- Bad for OLTP / Normal queries
- For a small number of analytics queries, the more CPUs the better
- Sometimes we run many shards on one server! (Microsharding)
  This works where:
  - All data can fit in RAM - Disk is not a bottleneck
  - Small number of power users - more CPUs than users
  - Write aggregations that can use parallelism

Clarification on (b) more CPUs than users:

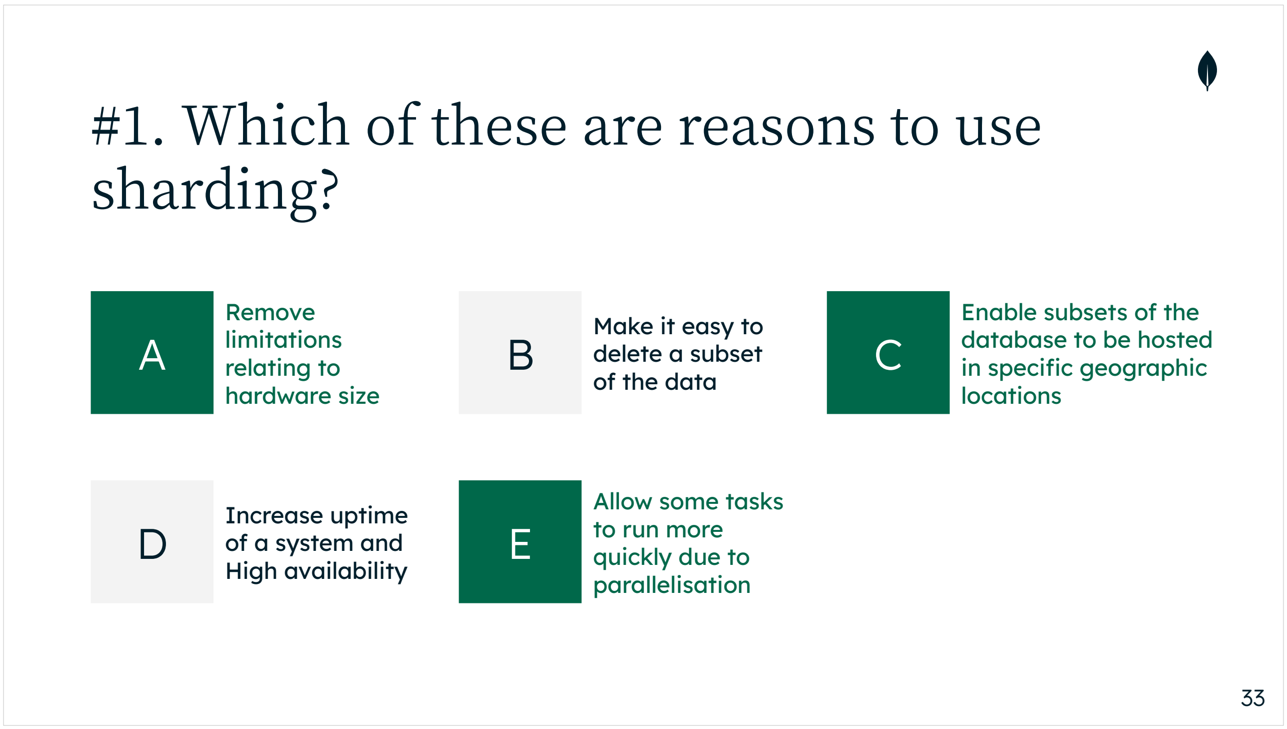
- The query engine doesn't parallelize a single query, so if a query is perform on a non-sharded collection, it will effectively use one CPU core out of potentially many.
- If a collection is queried at a high rate, using microsharding effectively allows users to use more than one core for one query,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

Quiz Time!

#1. Which of these are reasons to use sharding?

A	Remove limitations relating to hardware size	B	Make it easy to delete a subset of the data	C	Enable subsets of the database to be hosted in specific geographic locations
D	Increase uptime of a system and High availability	E	Allow some tasks to run more quickly due to parallelisation		

Answer in the next slide.



#1. Which of these are reasons to use sharding?

- [✓ A] Remove limitations relating to hardware size
- B Make it easy to delete a subset of the data
- [✓ C] Enable subsets of the database to be hosted in specific geographic locations
- D Increase uptime of a system and High availability
- [✓ E] Allow some tasks to run more quickly due to parallelisation

Sharding does not increase system update as replication provided by replica sets do that already. Have multiple shards would also mean that delete data would have to be deleted from multiple places if that collection were to be sharded so B & D aren't acceptable answers here. We can use sharded clusters for to remove hardware constraints, organize data in specific geographic locations, and implement tasks in parallel due to having additional replica sets.

#2. What are the things to do when loading a large data set into MongoDB.

A Presplit and move key ranges in the collection

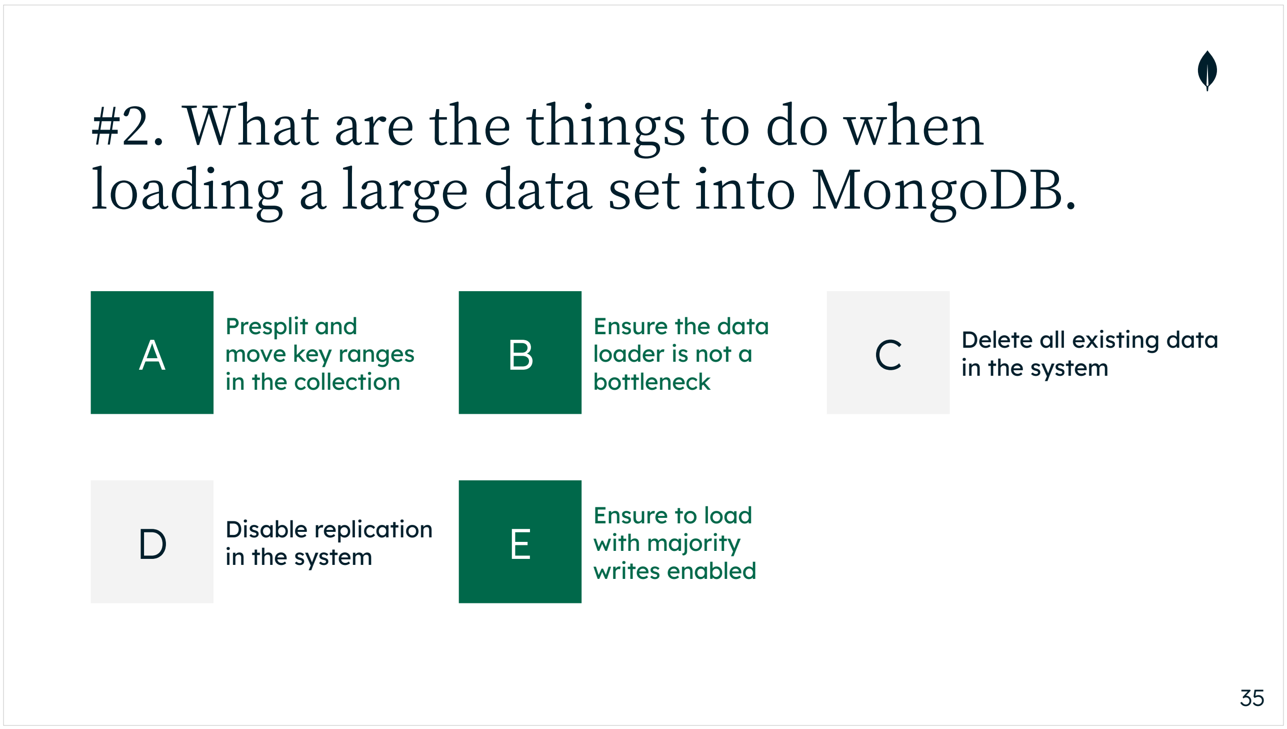
B Ensure the data loader is not a bottleneck

C Delete all existing data in the system

D Disable replication in the system

E Ensure to load with majority writes enabled

Answer in the next slide.



#2. What are the things to do when loading a large data set into MongoDB.

[✓ A] Presplit and move key ranges in the collection

[✓ B] Ensure the data loader is not a bottleneck

C Delete all existing data in the system

D Disable replication in the system

[✓ E] Ensure to load with majority writes enabled

Ensuring that writes are loaded with majority write concern is big as we wouldn't want to lose data in the event the operation fails part way. Single thread data loader would also make loading data much slower since we have multiple replica sets to work with in a sharded cluster and to further optimize the movement of the documents across various key ranges - it is highly recommended to presplit the data so the load balancer doesn't have to constantly rebalance as the data is being loaded.

Recap

Sharding requires optimized indexes and schema design to be effective.

Sharding can be implemented to remove hardware limitations, to host subsets in different geographical locations and allow parallelisation for certain tasks to occur.

Manually balancing data will be more performant over leaving it to the balancer to perform the same task when initially loading dataset.

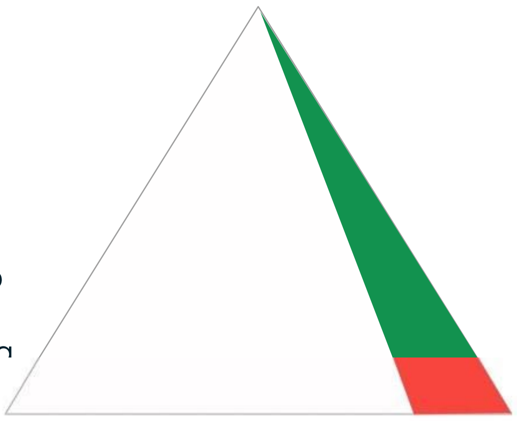
Appendix

Downsides of hashed indexes

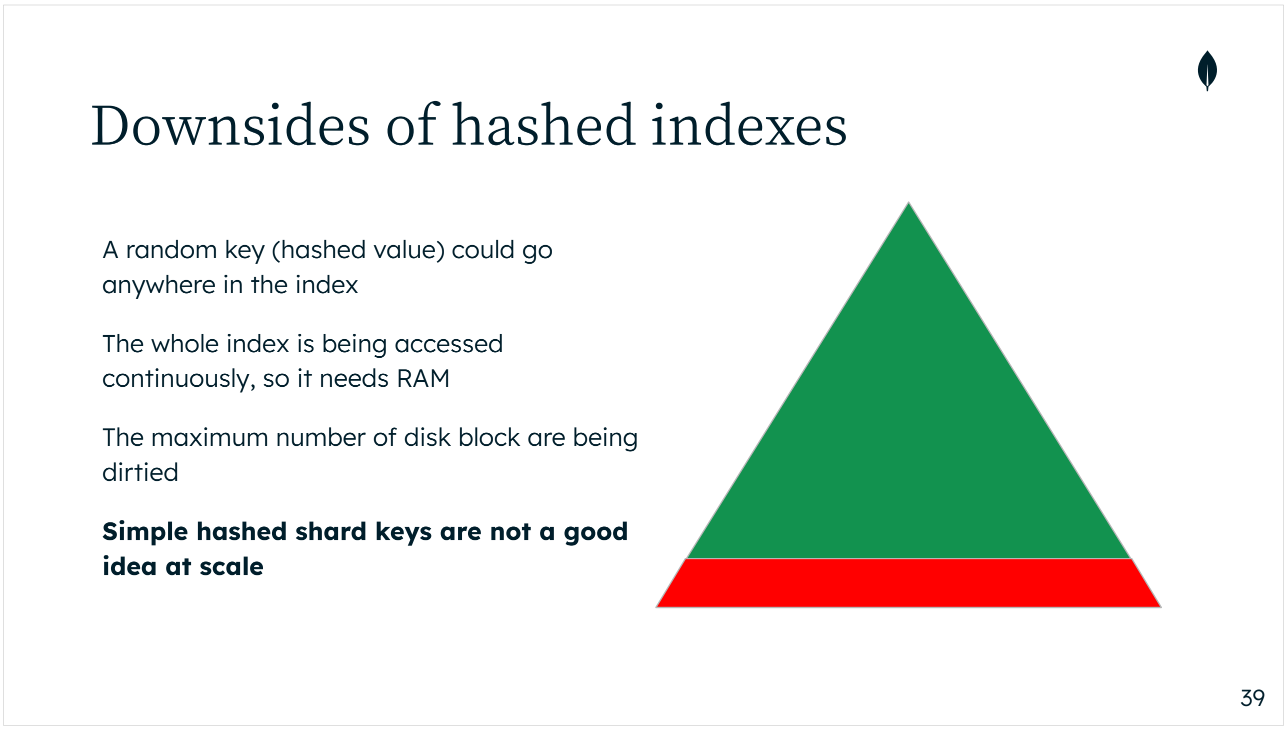
Indexes are BTrees with internal and leaf nodes

- 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



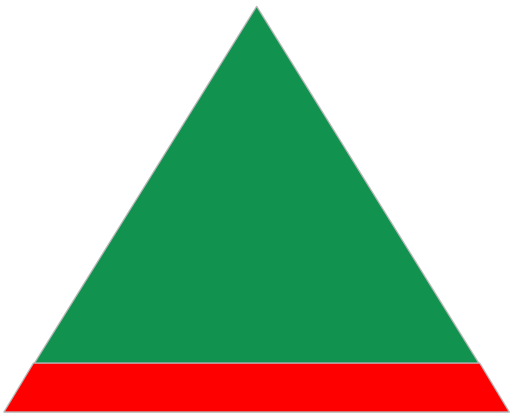
Downsides of hashed indexes

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

Simple hashed shard keys are not a good idea at scale



This also applies to other random values like GUIDs in indexes