MongoDB Sharding

CS185C: Introduction to NoSQL Databases

Suneuy Kim

Vertical Scaling vs. Horizontal Scaling

- Vertical Scaling
  - Increasing the capacity of a single server by adding more RAM, using a more powerful CPU, or increasing the amount of storage space.
  - There is a practical maximum for vertical scaling
- Horizontal Scaling
  - Divides the system dataset and load over multiple servers, adding additional servers to increase capacity
  - Increases complexity in infrastructure and maintenance.
- MongoDB supports horizontal scaling through sharding.

When to shard

- To increase available RAM
- To increase available disk space
- To reduce load on a server
- To read or write data with greater throughput than a single mongod can handle

Sharding (= horizontal partitioning)

- The process of splitting up a large collection, i.e. containing a large number of documents, across multiple independent servers to improve performance.
- MongoDB implements sharding at the collection level, not the database level.
- MongoDB adopts auto-sharding
  - A cloud of shards can be treated as a single logical database.
  - An application is not aware of that the data are distributed across multiple servers.

Requirements of Sharding System

1. The ability to distribute data evenly across all shards - balancer
2. The ability to store shard data in a fault-tolerant fashion -

   The ability to use replica sets as a storage mechanism for shards

3. The ability to add or remove shards while the system is running – config server

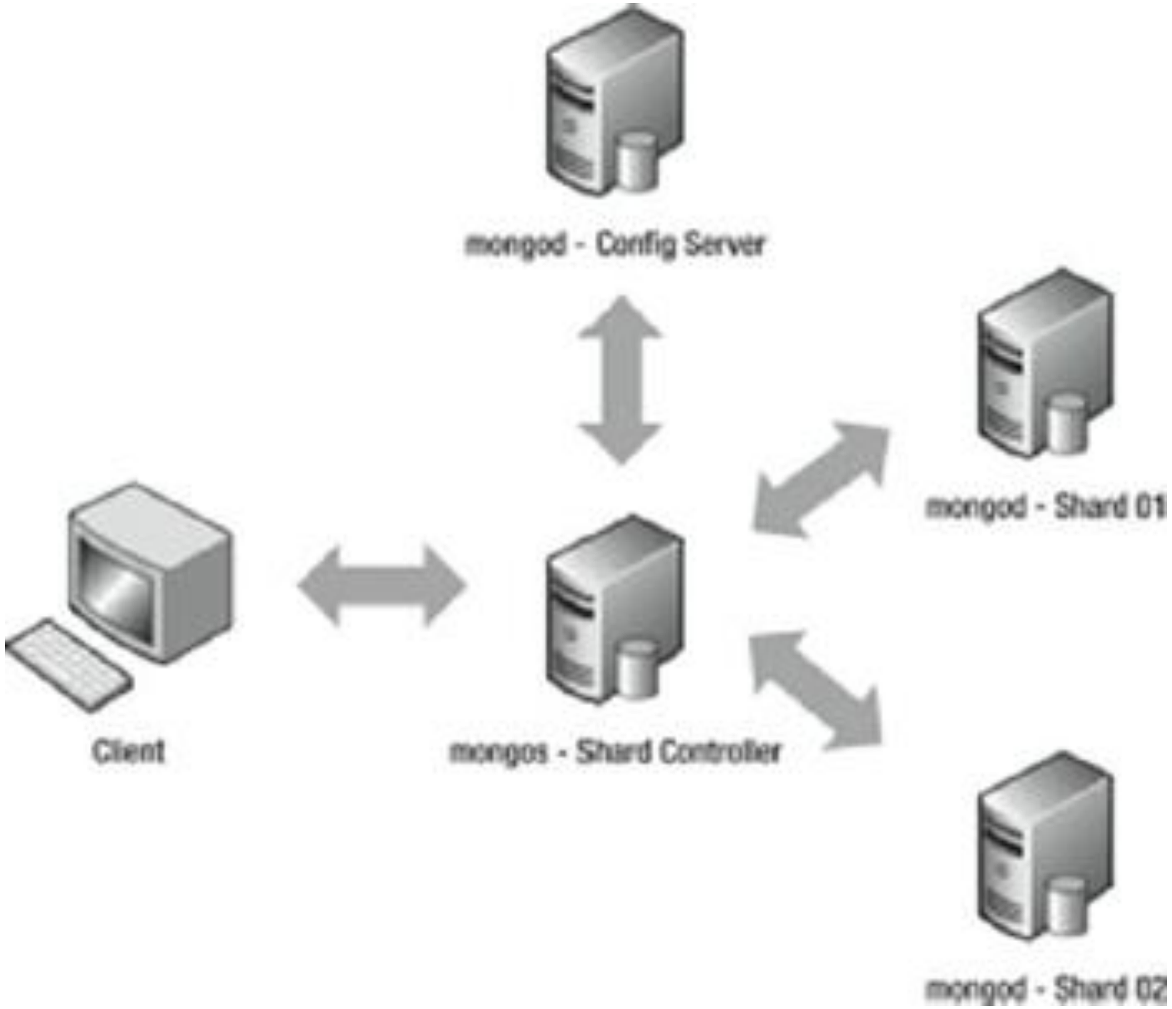
Sharded Cluster Components

- Shard
  - Each shard contains a subset of the sharded data.
  - Each shard can be deployed as a replica set.
- Config servers
  - A config server, which is a `mongod` server, acts as a directory that allows the location of each chunk to be determined.
  - For this, a config server stores metadata and configuration settings for the cluster.
  - As of MongoDB 3.2, config servers can be deployed as a replica set. (3 config servers are recommended.)
- mongos – next page

mongos

- The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
- A `mongos` routing process manages
  1) splitting of data (splitting chunks)
  2) routing of requests to the required shard server
  3) merging the data from multiple shards to answer a query
- Applications connect to the mongos and issue requests to it.
- The shard key maps data into chunks.
- Chunks are logical contiguous ranges of document keys.
  - Each chunk identifies a number of documents with a particular continuous range of sharding key values.

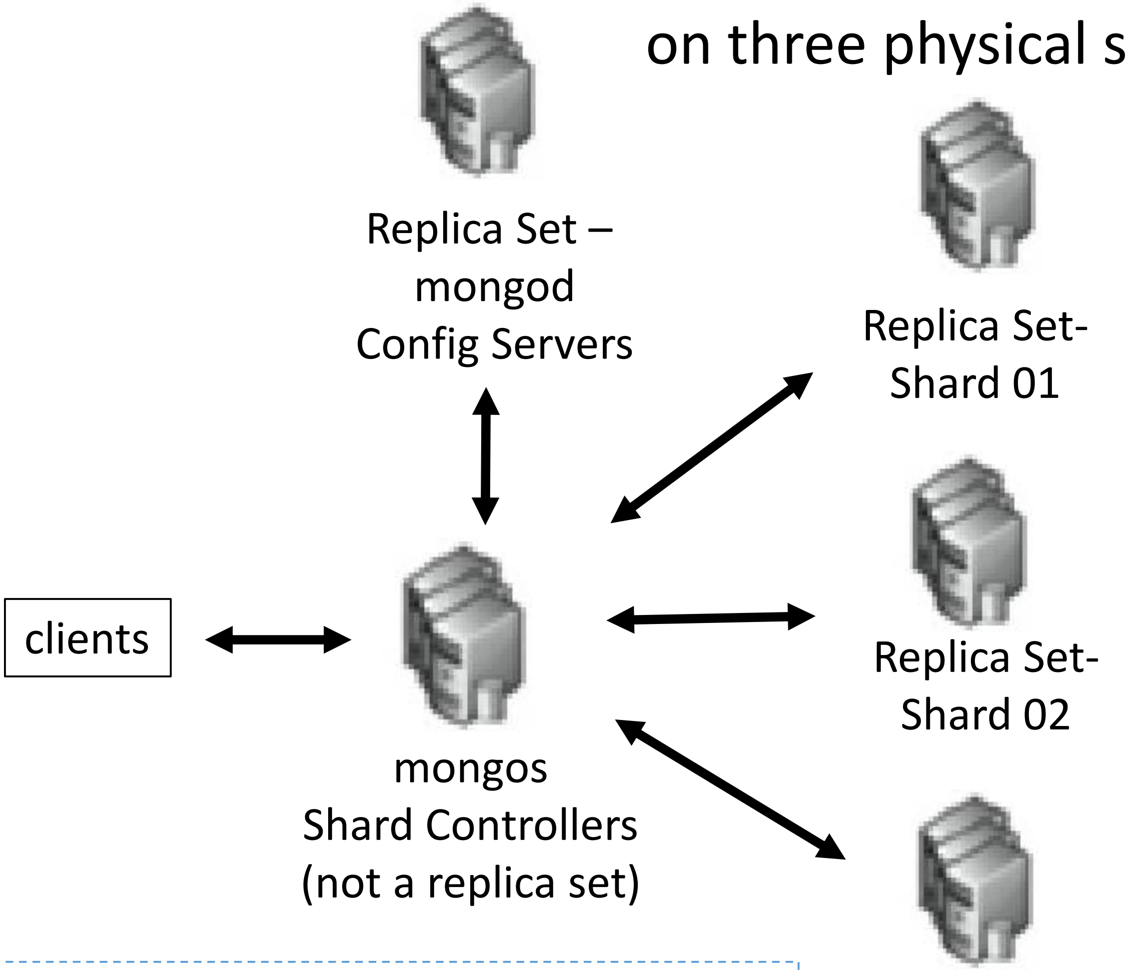
Simple sharding setup without redundancy

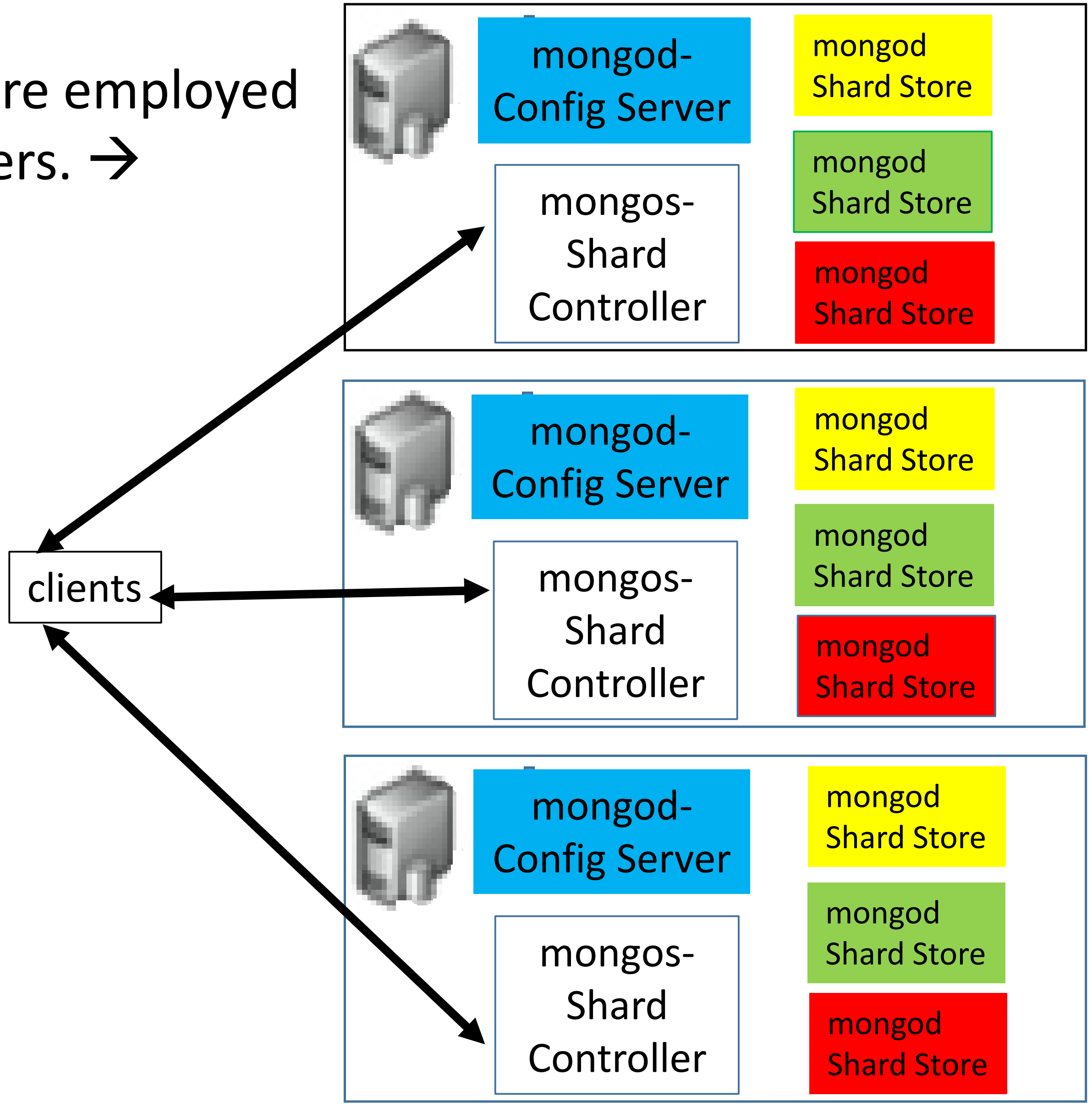


The instruction for this set up is presented in a separate document titled “sharding configuration on single machine”.

A redundant sharding configuration

When these services are employed on three physical servers. →


Replica Set – mongod Config Servers
clients
mongos Shard Controllers (not a replica set)
Replica Set- Shard 01
Replica Set- Shard 02
Replica Set- Shard 03



Common setup: One mongos per application server running on the same machine as the application server.

Sharding Data

- You must explicitly tell both the database and collection that you want them to be distributed.
- Sharding database is always prerequisite to sharding one of its collections

To shard the artists collection in the music database on the "name" key

```
>db.enableSharding("music")
>sh.shardCollection("mustic.artists", {"name":1})
```

- The shardCollection command splits the collection into chunks based on the value of the shard key, name in the above example.
- For an existing collection, there must be an index on the shard key. Otherwise, an error occurs.
- For a non-existing collection, an index on the shard key is automatically created.

Chunks

- The unit MongoDB uses to move data around
- A group of documents in a given range of the shard key
- A chunk always lives on a single shard.
- Once a chunk grows to a certain size, due to insertions, MongoDB automatically splits it into two smaller chunks.
- Chunks are exclusive without any overlapping ranges.
- A document belongs one and only one chunk → an array field cannot be used as a shard key.
- A chunk is not necessarily grouped together on disk.

Chunk Ranges

- A newly sharded collection starts off with a single chunk.
- As a chunk grows, it is automatically split into two chunks.

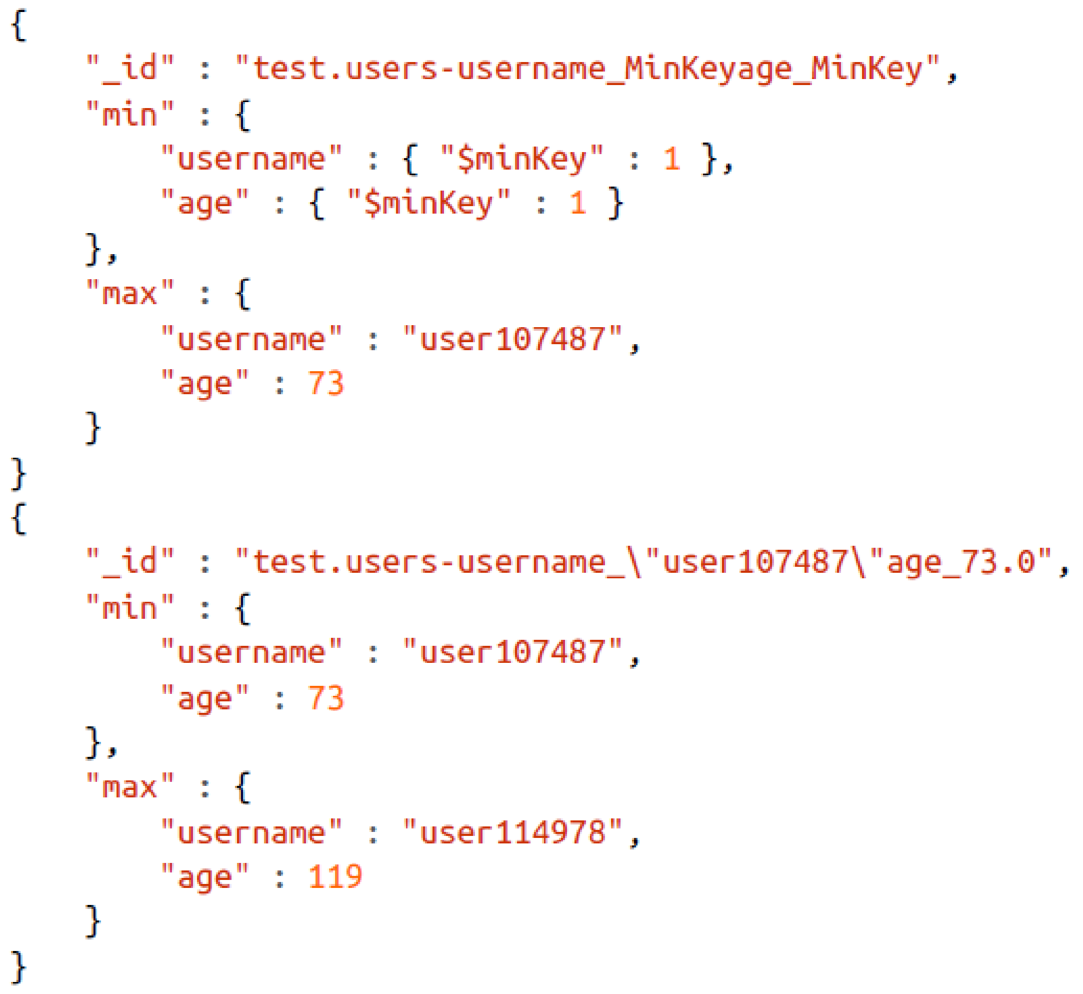
Example: A chunk with a range 3<= age < 17 splits into two ranges: 3<=age < 12 on one chunk and 12<=age < 17 on the other. 12 is the split point.

- Assignment: compound sharding key

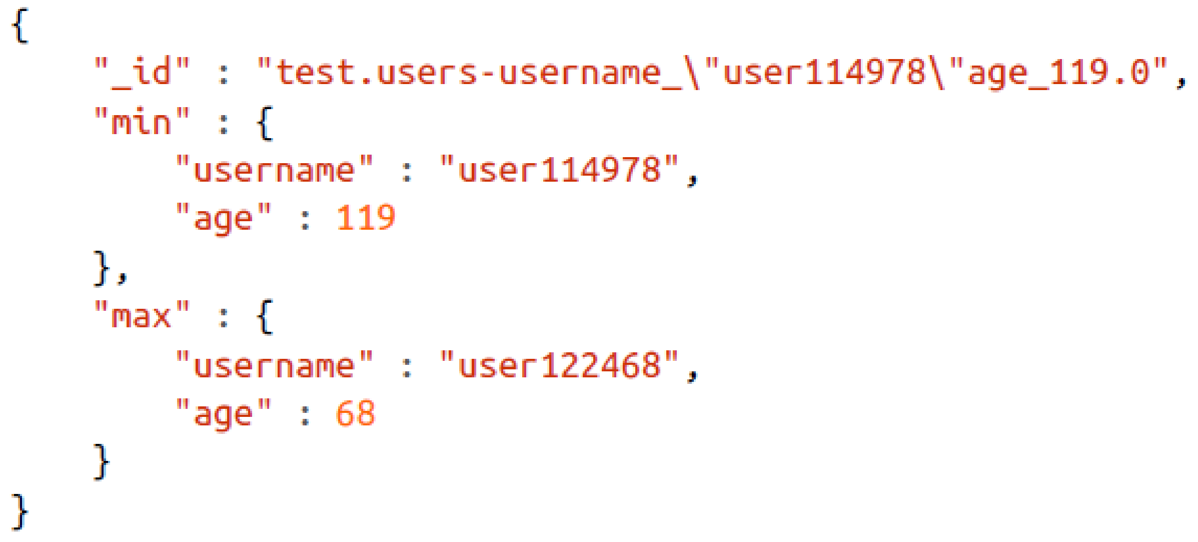
1) Setup a sharded collection with 2 shards using a compound shard key on (“username”:1, “age”:1). Use chunk size 1.
2) Populate this collection documents consisting of username and age.
3) Show the chunk information of this collection
4) Elaborate what type of query can benefits from this sharded collection.

e.g. To find on which chunk someone with a given user name vs.
To find one which chunk someone with a given age

With a compound shard key on (“username”:1, “age”:1)



```
{
    "_id" : "test.users-username_MinKeyage_MinKey",
    "min" : {
        "username" : { "$minKey" : 1 },
        "age" : { "$minKey" : 1 }
    },
    "max" : {
        "username" : "user107487",
        "age" : 73
    }
}
{
    "_id" : "test.users-username_\"user107487\"age_73.0",
    "min" : {
        "username" : "user107487",
        "age" : 73
    },
    "max" : {
        "username" : "user114978",
        "age" : 119
    }
}
```



```
{
    "_id" : "test.users-username_\"user114978\"age_119.0",
    "min" : {
        "username" : "user114978",
        "age" : 119
    },
    "max" : {
        "username" : "user122468",
        "age" : 68
    }
}
```

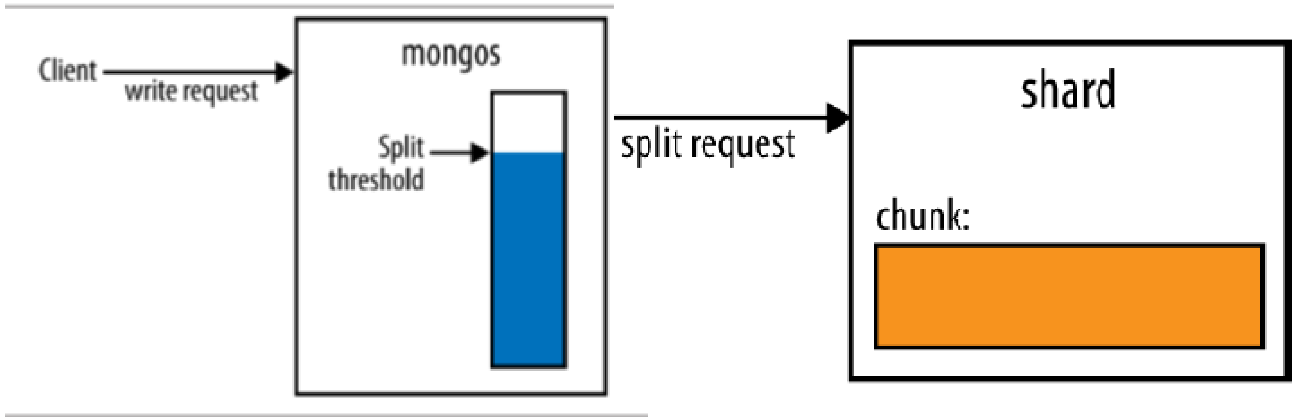
Chunk Ranges

- Chunk information is stored in the config.chunks collection.

```
vagrant@vagrant-ubuntu-trusty-64:~$ sudo mongo 127.0.0.1:27021
mongos> use config
mongos> db.chunks.find()
{ "_id" : "testdb.testcollection-testkey_MinKey", "lastmod" : Timestamp(10, 0),
"lastmodEpoch" : ObjectId("5858a1e8b4ccc8eb41074bca"), "ns" : "testdb.testcollection",
"min" : { "testkey" : { "$minKey" : 1 } }, "max" : { "testkey" : "key0" }, "shard" : "shard0000" }
{ "_id" : "testdb.testcollection-testkey_\"key0\"", "lastmod" : Timestamp(8, 1),
"lastmodEpoch" : ObjectId("5858a1e8b4ccc8eb41074bca"), "ns" : "testdb.testcollection",
"min" : { "testkey" : "key0" }, "max" : { "testkey" : "key19970" }, "shard" : "shard0000" }
```

- The complete result can be found “chunk information” document on the course web site.

Splitting Chunks



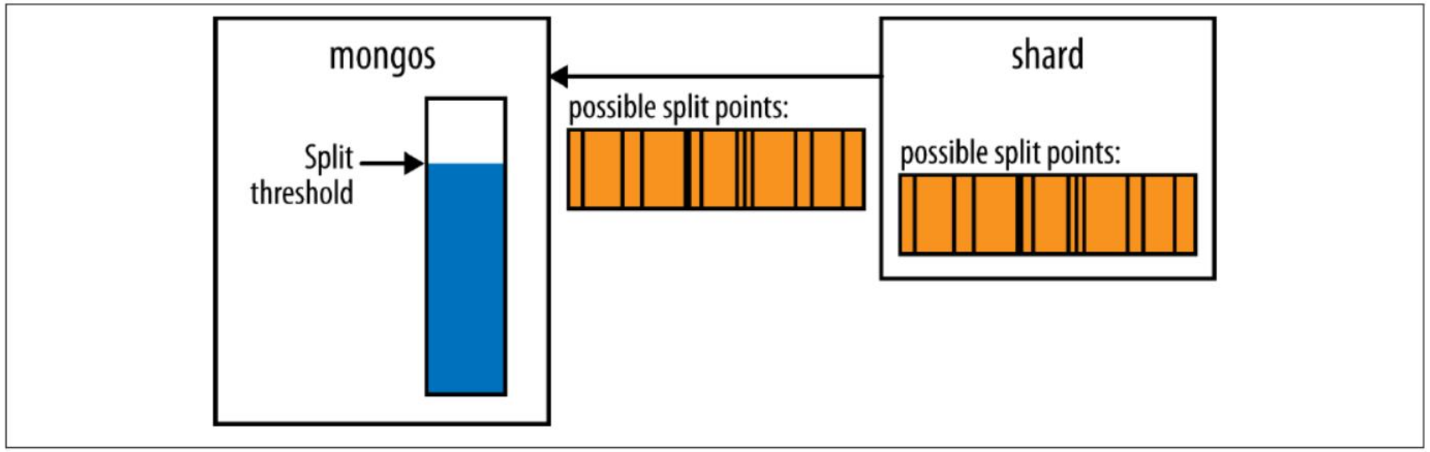
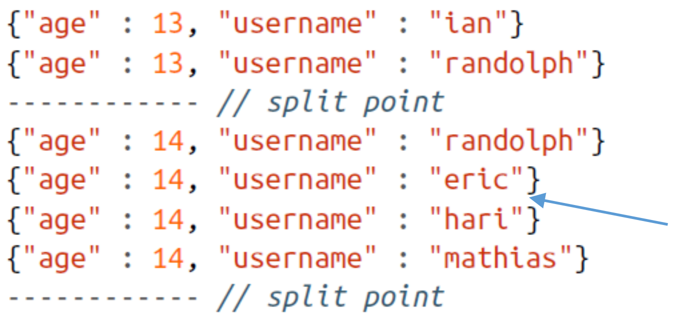


Figure 14-3. The shard calculates split points for the chunk and sends them to the mongos

- monogs tracks how much data it writes per chunk and
- If the split threshold is reached, mongos will send a request for split points to the shard.
- The shard calculate split points for the chunk and sends them to the mongos

Splitting chunks

- Chunk splits are just a metadata change (no data move) in the config server.
- After new chunk documents are created on the config servers and the old chunk's range ("max") is modified, the mongos resets its tracking for the original chunk and creates new trackers for the new chunk
- Chunks can only be split between documents where the shard key's value changes – documents with the same shard key must live in the same chunk



```
{"age" : 13, "username" : "ian"}
{"age" : 13, "username" : "randolph"}
------------ // split point
{"age" : 14, "username" : "randolph"}
{"age" : 14, "username" : "eric"}
{"age" : 14, "username" : "hari"}
{"age" : 14, "username" : "mathias"}
------------ // split point
```

cannot split here
if only age is the shard key.

- Having a variety values for a shard key is important.

e.g. What if "state" is a shard key?

Split storm

- All config servers must be up for splits to happen.
- If a config server is down when a mongos tries to do a split, the mongos cannot update the metadata and the split fails.
- Ensure your config servers are up and healthy not to have a split storm.

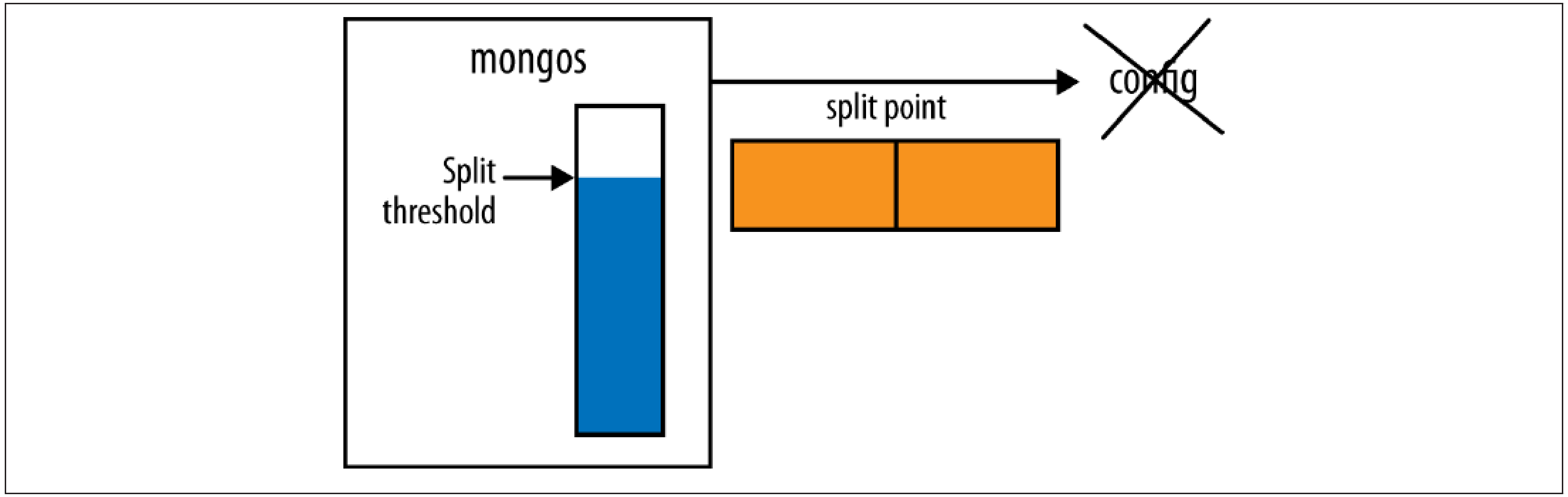


Figure 14-4. The mongos chooses a split point and attempts to inform the config server but cannot reach it. Thus, it is still over its split threshold for the chunk and any subsequent writes will trigger this process again.

Case where chunks grow without bound

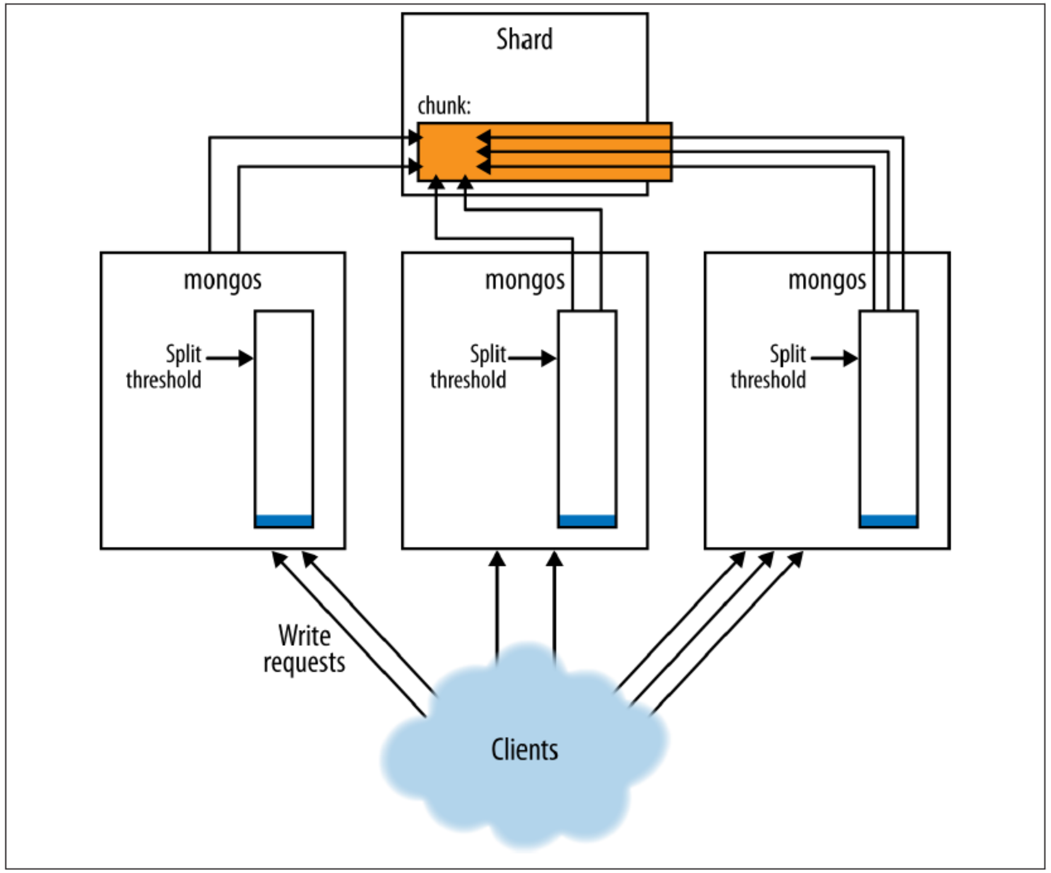


Figure 14-6. If mongos processes are regularly restarted their counters may never hit the threshold, making chunks grow without bound

1. Leave mongos processes up.
2. Make a chunk size smaller in a way that mongos prompts splits to happen at a lower threshold.

Config servers

- Config servers store the metadata for a sharded cluster.
- The metadata includes the list of chunks on every shard and the ranges that define the chunks.
- Config servers must be set up first and the metadata they hold is extremely important.
- Three config servers are recommended.
- The mongos instances cache this data and use it to route read and write operations to the correct shards.
- A mongos reads data from the config server in the following cases:
    - A new mongos starts for the first time, or an existing mongos restarts.
    - After change in the cluster metadata, such as after a chunk migration.
- A mongos only writes data to the config servers when the metadata changes, such as
    - after a chunk migration or after a chunk split

The balancer

- As an element of mongos process, it checks its table of chunks for each collection to see if any shards have hit the balancing threshold.
- If an imbalance is detected, the balancer redistributes chunks within a cluster (migration) to ensure even distribution of data among shards.
- Migration in progress is not perceived by an application. All reads and writes are routed to the old chunk until the migration is complete.
- Once migration is done, all mongos attempting to access the data in the old location will get an error. If then, mongos looks up the new location of the data from config servers, updates its chunk table, and tries the request again.

The balancer

```
mongos> sh.stopBalancer()
Waiting for active hosts...
Waiting for the balancer lock...
Waiting again for active hosts after balancer is off...
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

mongos> db.settings.find({_id:"balancer"})
{ "_id" : "balancer", "stopped" : true }

mongos> sh.startBalancer()
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Choosing a shard key

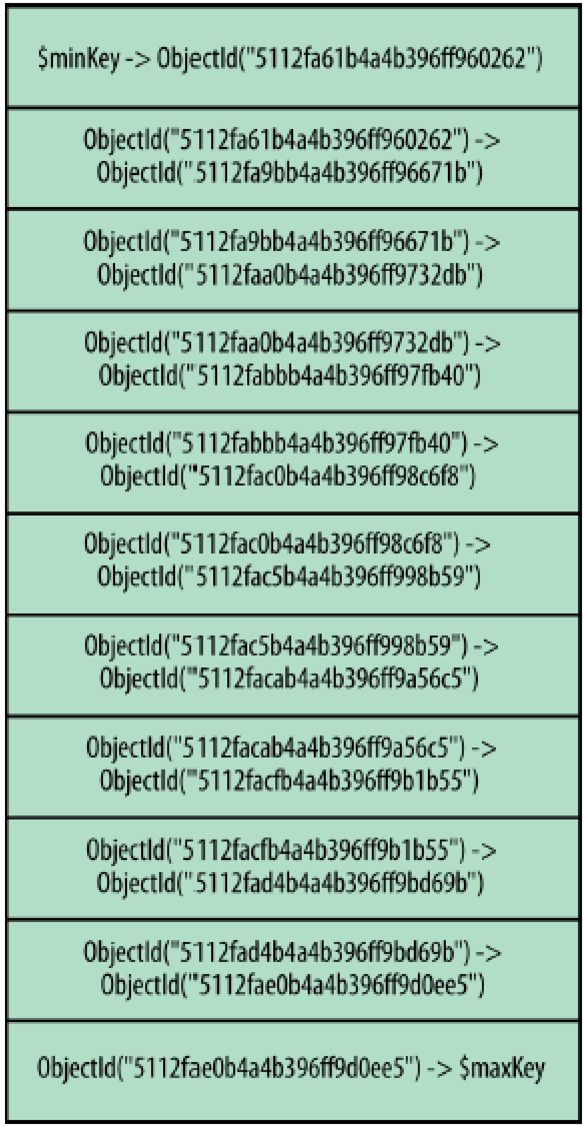
- Shard key: a field or two to use to split up the data
- Once you set a shard key, it cannot be changed.
- MongoDB's efficient data storage, scaling, and querying depend on sharding, and sharding depends on the careful selection of a shard key.
    - Query isolation: A shard key that is tied to a single shard
    - Write scaling: When a field has a high degree of randomness
- Questions to ask to choose a shard key
    - How many shards are you planning to grow to?
    - Are you sharding to reduce read/write latency?
    - Are you sharding to increase read/write throughput?
    - Are you sharding to increase system resources?

Choosing a shard key

- Shard key classification based on the nature of key values
  - ascending key
  - random key
  - location-based key
- Sharding strategies
  - range-based sharding
  - hash-based sharding
  - tag aware sharding

Ascending Shard Keys

- A key that steadily increases over time. e.g. ObjectId
- Max chunk: the chunk containing $maxKey
- With this pattern, all the newly added documents will be in the max chunk.
  - All of your writes will be routed to one shard containing the max chunk.
  - The max chunk continues growing and being split into multiple chunks.
- All the chunks are being created by one shard – difficult for MongoDB to keep chunks evenly balanced.



$minKey -> ObjectId("5112fa61b4a4b396ff960262")

ObjectId("5112fa61b4a4b396ff960262") -> ObjectId(".5112fa9bb4a4b396ff96671b")

ObjectId("5112fa9bb4a4b396ff96671b") -> ObjectId("5112faa0b4a4b396ff9732db")

ObjectId("5112faa0b4a4b396ff9732db") -> ObjectId("5112fabbb4a4b396ff97fb40")

ObjectId("5112fabbb4a4b396ff97fb40") -> ObjectId("5112fac0b4a4b396ff98c6f8")

ObjectId("5112fac0b4a4b396ff98c6f8") -> ObjectId("5112fac5b4a4b396ff998b59")

ObjectId("5112fac5b4a4b396ff998b59") -> ObjectId('"5112facab4a4b396ff9a56c5")

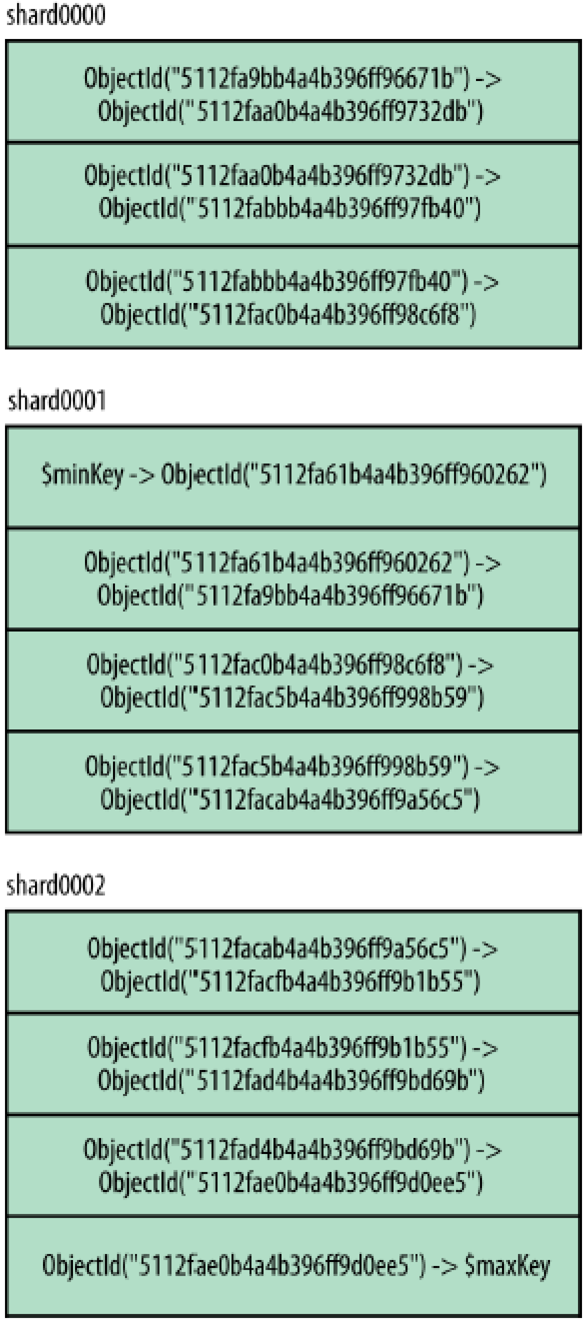
ObjectId("5112facab4a4b396ff9a56c5") -> ObjectId('"5112facfb4a4b396ff9b1b55")

ObjectId("5112facfb4a4b396ff9b1b55") -> ObjectId("5112fad4b4a4b396ff9bd69b")

ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey

The collection is split into ranges of ObjectIds. Each range is a chunk.



shard0000

ObjectId("5112fa9bb4a4b396ff96671b") -> ObjectId("5112faa0b4a4b396ff9732db")

ObjectId("5112faa0b4a4b396ff9732db") -> ObjectId("5112fabbb4a4b396ff97fb40")

ObjectId("5112fabbb4a4b396ff97fb40") -> ObjectId("5112fac0b4a4b396ff98c6f8")

shard0001

$minKey -> ObjectId("5112fa61b4a4b396ff960262")

ObjectId("5112fa61b4a4b396ff960262") -> ObjectId("5112fa9bb4a4b396ff96671b")

ObjectId("5112fac0b4a4b396ff98c6f8") -> ObjectId("5112fac5b4a4b396ff998b59")

ObjectId("5112fac5b4a4b396ff998b59") -> ObjectId("5112facab4a4b396ff9a56c5")

shard0002

ObjectId("5112facab4a4b396ff9a56c5") -> ObjectId("5112facfb4a4b396ff9b1b55")

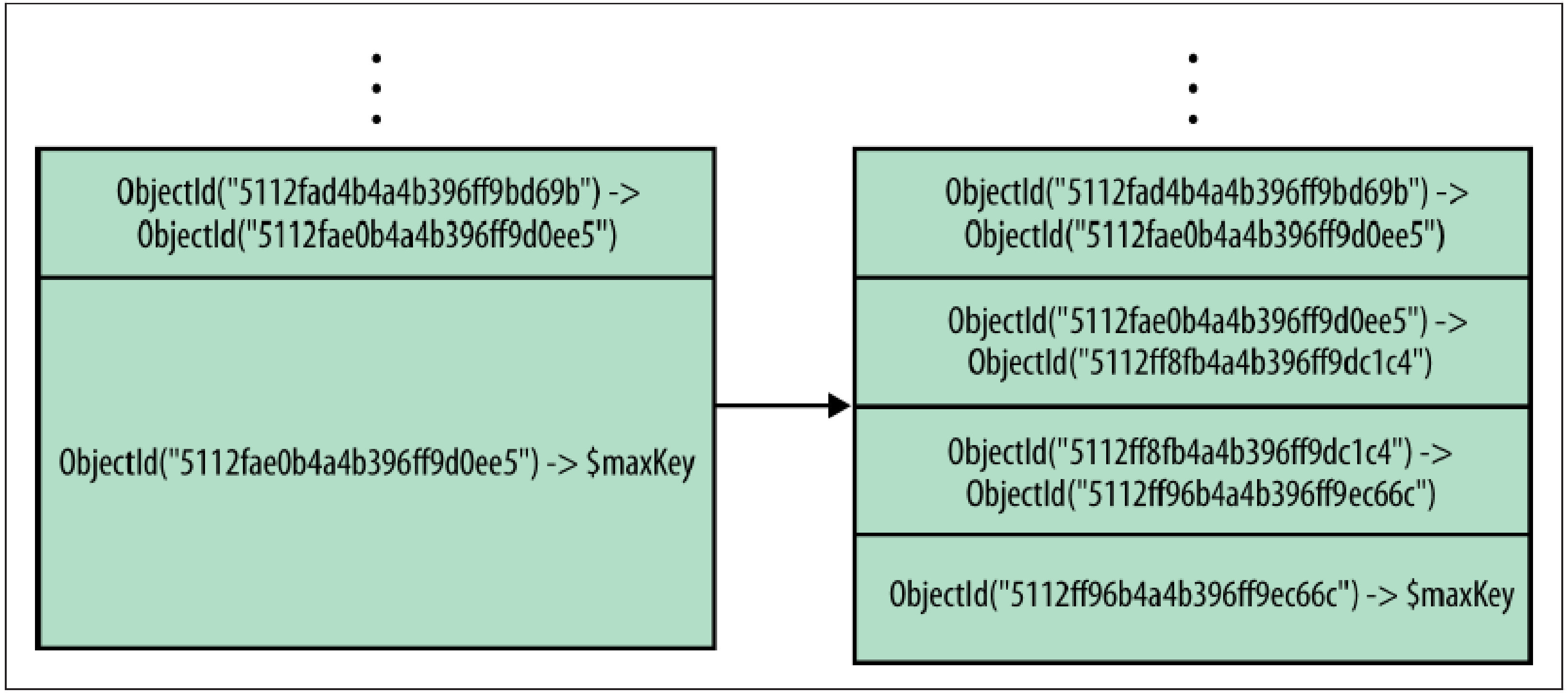
ObjectId("5112facfb4a4b396ff9b1b55") -> ObjectId("5112fad4b4a4b396ff9bd69b")

ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey

Chunks are distributed across shards in a random order

Ascending Shard Keys



ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey

ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> ObjectId("5112ff8fb4a4b396ff9dc1c4")

ObjectId("5112ff8fb4a4b396ff9dc1c4") -> ObjectId("5112ff96b4a4b396ff9ec66c")

ObjectId("5112ff96b4a4b396ff9ec66c") -> $maxKey

Figure 15-3. The max chunk continues growing and being split into multiple chunks

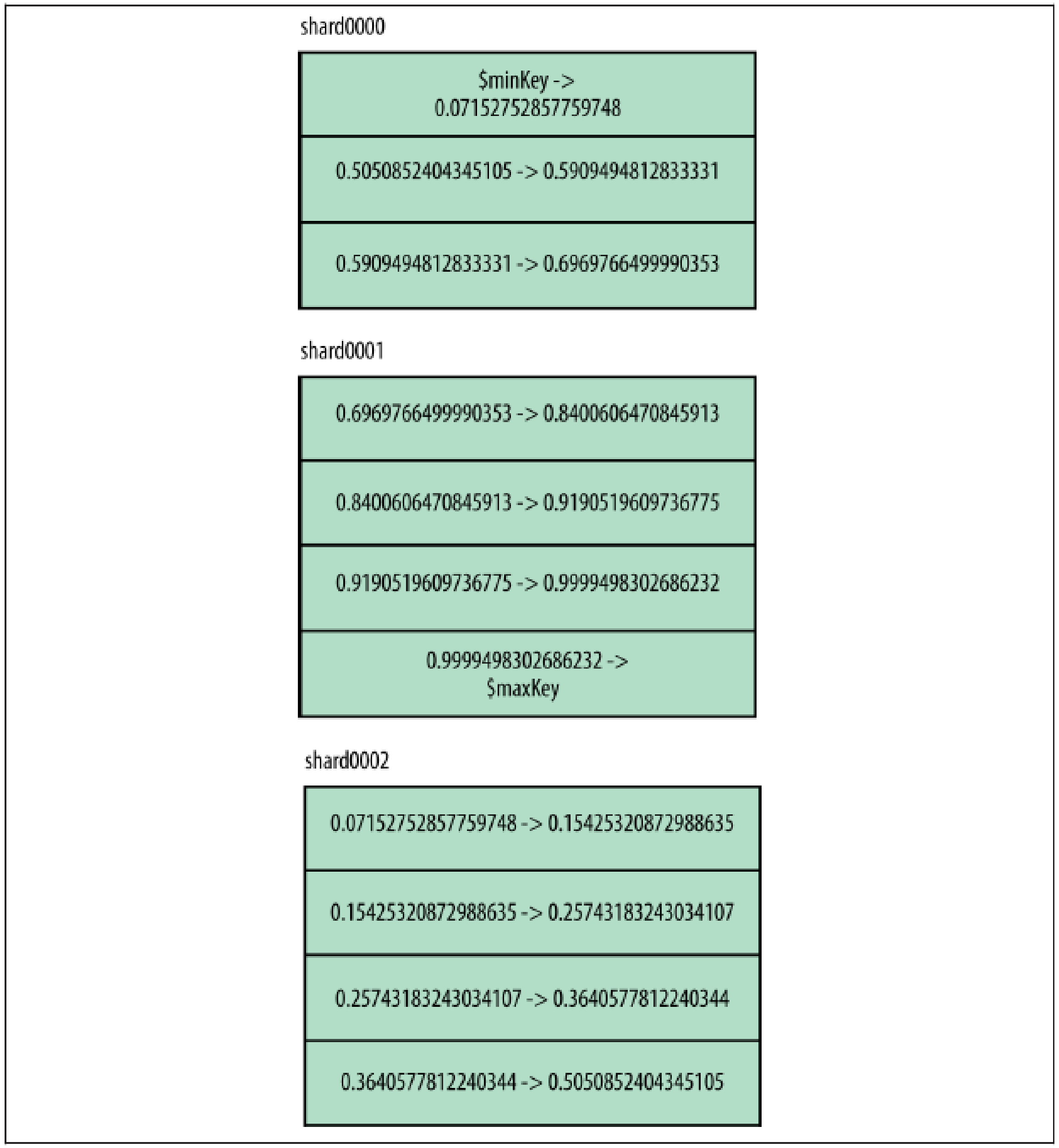
Randomly Distributed Shard Keys

- Keys that has no identifiable pattern in your dataset.

Examples: Usernames, email addresses, UUID, MD5 hashes, etc.

- Pro: Inserts should hit every chunk fairly evenly and shards grow roughly the same rate → the number of migrates will be limited.
- Con: MongoDB is not efficient at randomly accessing data beyond the size of RAM.

Randomly Distributed Shard Keys



shard0000

$minKey -> 0.07152752857759748

0.5050852404345105 -> 0.5909494812833331

0.5909494812833331 -> 0.6969766499990353

shard0001

0.6969766499990353 -> 0.8400606470845913

0.8400606470845913 -> 0.9190519609736775

0.9190519609736775 -> 0.9999498302686232

0.9999498302686232 -> $maxKey

shard0002

0.07152752857759748 -> 0.15425320872988635

0.15425320872988635 -> 0.25743183243034107

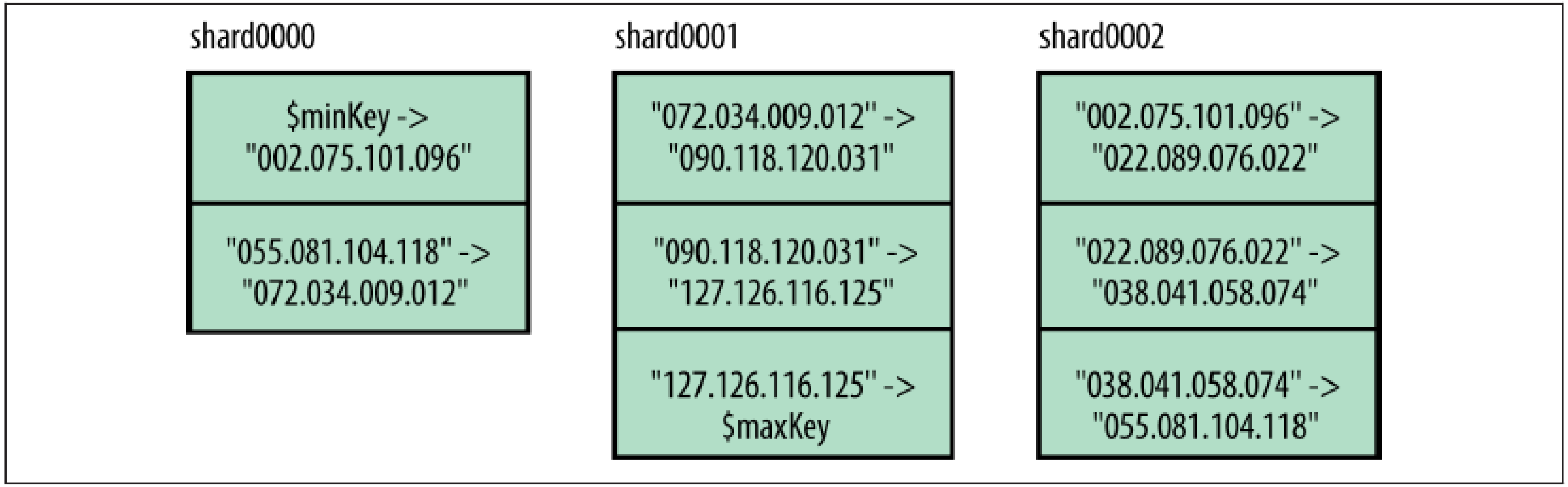
0.25743183243034107 -> 0.3640577812240344

0.3640577812240344 -> 0.5050852404345105

Figure 15-4. As in the previous section, chunks are distributed randomly around the cluster

Location-Based Shard Keys

- A location-based key is a key that makes documents with some similarity fall into a range based on this field.
- A location does not necessarily mean a physical location field
- Example: user’s IP, latitude, and longitude, or address



shard0000	shard0001	shard0002
$minKey -> "002.075.101.096"	"072.034.009.012" -> "090.118.120.031"	"002.075.101.096" -> "022.089.076.022"
"055.081.104.118" -> "072.034.009.012"	"090.118.120.031" -> "127.126.116.125"	"022.089.076.022" -> "038.041.058.074"
	"127.126.116.125" -> $maxKey	"038.041.058.074" -> "055.081.104.118"

Figure 15-5. A sample distribution of chunks in the IP address collection

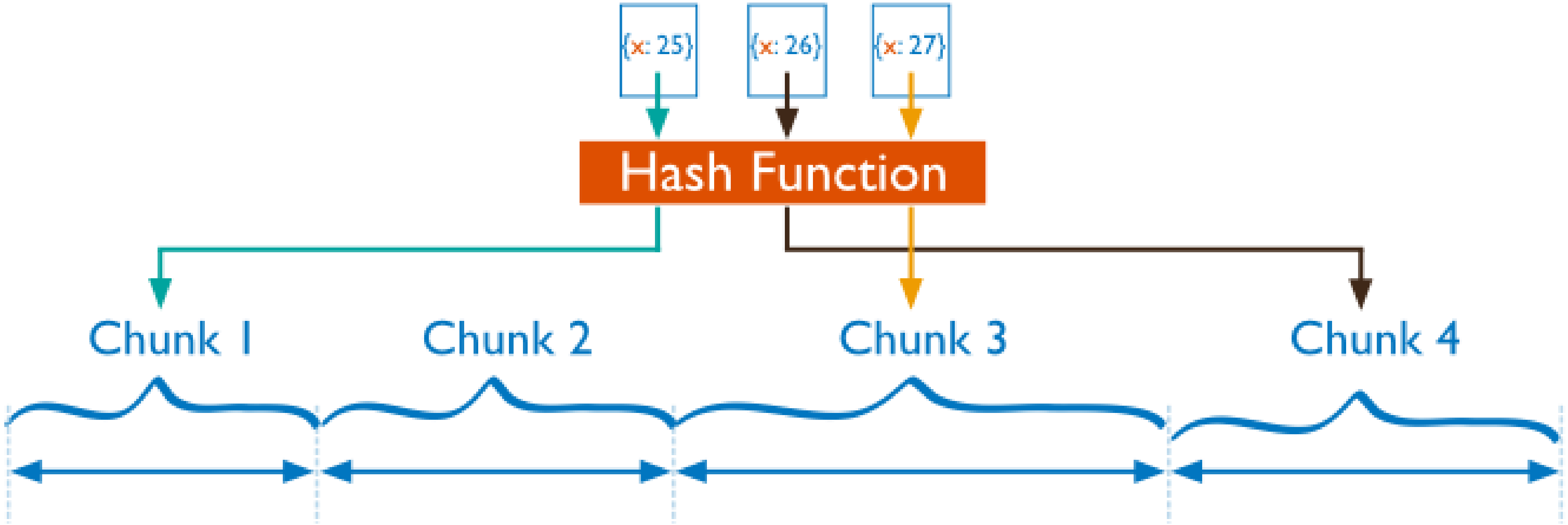
Shard Key Strategies

- MongoDB supports the sharding strategies for distributing data across sharded clusters.
  - Range-based sharding splits collections based on shard key value.
  - Hash-based sharding determines hash values based on field values in the shard key.
  - Tag aware sharding *tags* specific ranges of the *shard* key and associates those *tags* with a *shard* or subset of shards.

Hashed Sharding

- Hashed sharding uses a hashed index of a single field as the shard key to partition data across your sharded cluster.
- Each chunk is then assigned a range based on the hashed shard key values.
- Hashed sharding facilitates more even data distribution, especially in data sets with a ascending shard key.
- Limitations
  - Random data (and index) updates can be I/O intensive.
  - Range queries are costy. The mongos is more likely to perform Broadcast Operations to fulfill a given ranged query.

Hashed Sharding



To create a hashed index on the shard key

```
mongos> db.users.ensureIndex({"username":"hashed"})
{
        "raw" : {
                "127.0.0.1:27024" : {
                        "createdCollectionAutomatically" : true,
                        "numIndexesBefore" : 1,
                        "numIndexesAfter" : 2,
                        "ok" : 1
                }
        },
        "ok" : 1
}
```

Shard an empty collection testdb.users

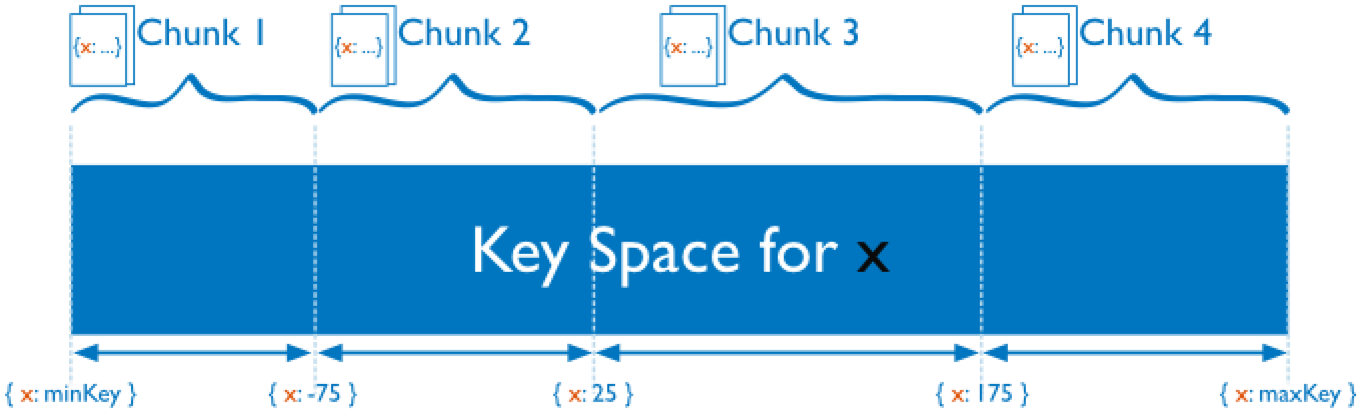
```
mongos> show databases
config  0.001GB
testdb  0.005GB
mongos> use testdb
switched to db testdb
mongos> show collections
testcollection
testtagsharding
users
mongos>sh.shardCollection("testdb.users",{"username":"hashed"})
{ "collectionsharded" : "testdb.users", "ok" : 1 }
```

When a hashed shard key is created on an empty collection, shardCollection creates two chunks per shard.

```
mongos> sh.status()
testdb.users
        shard key: { "username" : "hashed" } . . .
        chunks:  shard0000      2
                 shard0001      2
{ "username" : { "$minKey" : 1 } } -->> { "username" : NumberLong(“-4611686018427387902") } on : shard0000 Timestamp(2, 2)
        { "username" : NumberLong("-4611686018427387902") } -->> { "username" : NumberLong(0) } on : shard0000 Timestamp(2, 3)
        { "username" : NumberLong(0) } -->> { "username" : NumberLong("4611686018427387902") } on : shard0001 Timestamp(2, 4)
        { "username" : NumberLong("4611686018427387902") } -->> { "username" : { "$maxKey" : 1 } } on : shard0001 Timestamp(2, 5)
```

Range Sharding

- Ranged sharding involves dividing data into ranges based on the shard key values.
- Each chunk is then assigned a range based on the shard key values.
- A range of shard keys whose values are “close” are more likely to reside on the same chunk. This allows for targeted operations.



Shard Key Selection for Ranged Sharding

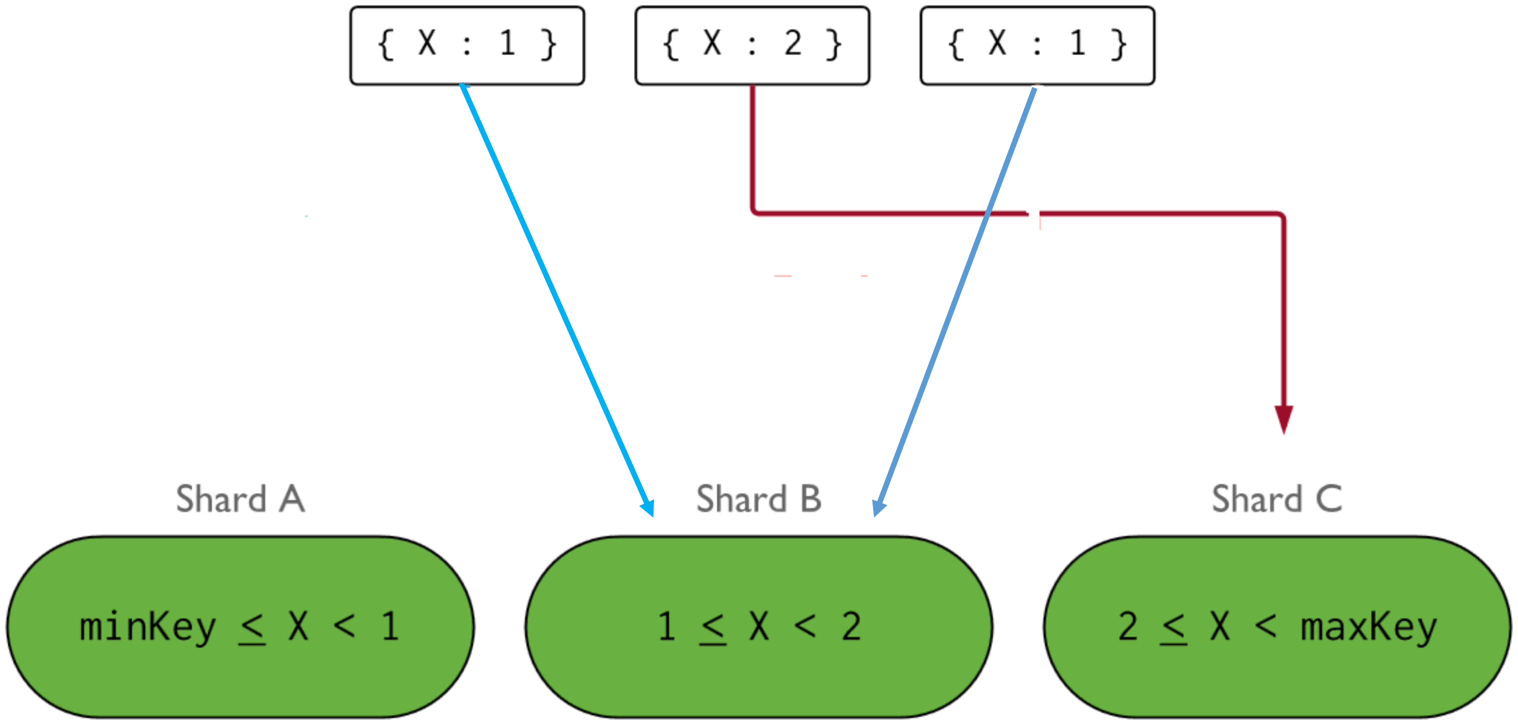
- Ranged sharding is most efficient when the shard key displays the following traits. Otherwise, the effectiveness of horizontal scaling will be reduced or removed.
  - Large Shard Key Cardinality
  - Low Shard Key Frequency
  - Non-Monotonically Changing Shard Keys

Shard key with low cardinality

- The cardinality of a shard key determines the maximum number of chunks the balancer can create.
- If a shard key has a cardinality of 4, then there can be no more than 4 chunks within the sharded cluster.
- This constrains the number of effective shards in the cluster to 4 as well - adding additional shards would not provide any benefit.

Shard key with low cardinality

The cluster in this example would *not* scale horizontally, as incoming writes would only route to a subset of shards.

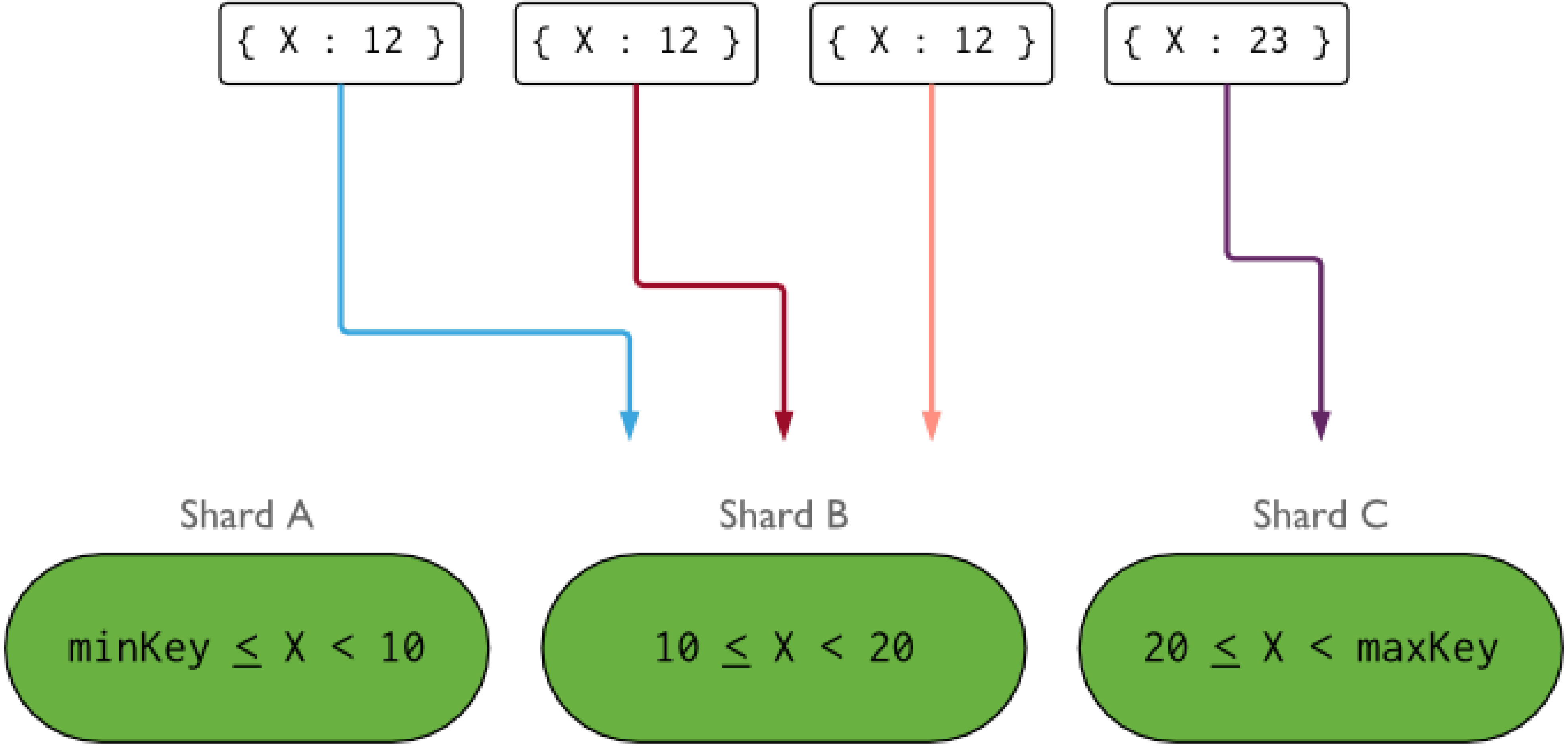


Adding more shards will not provides any benefit.

Shard Key with High Frequency

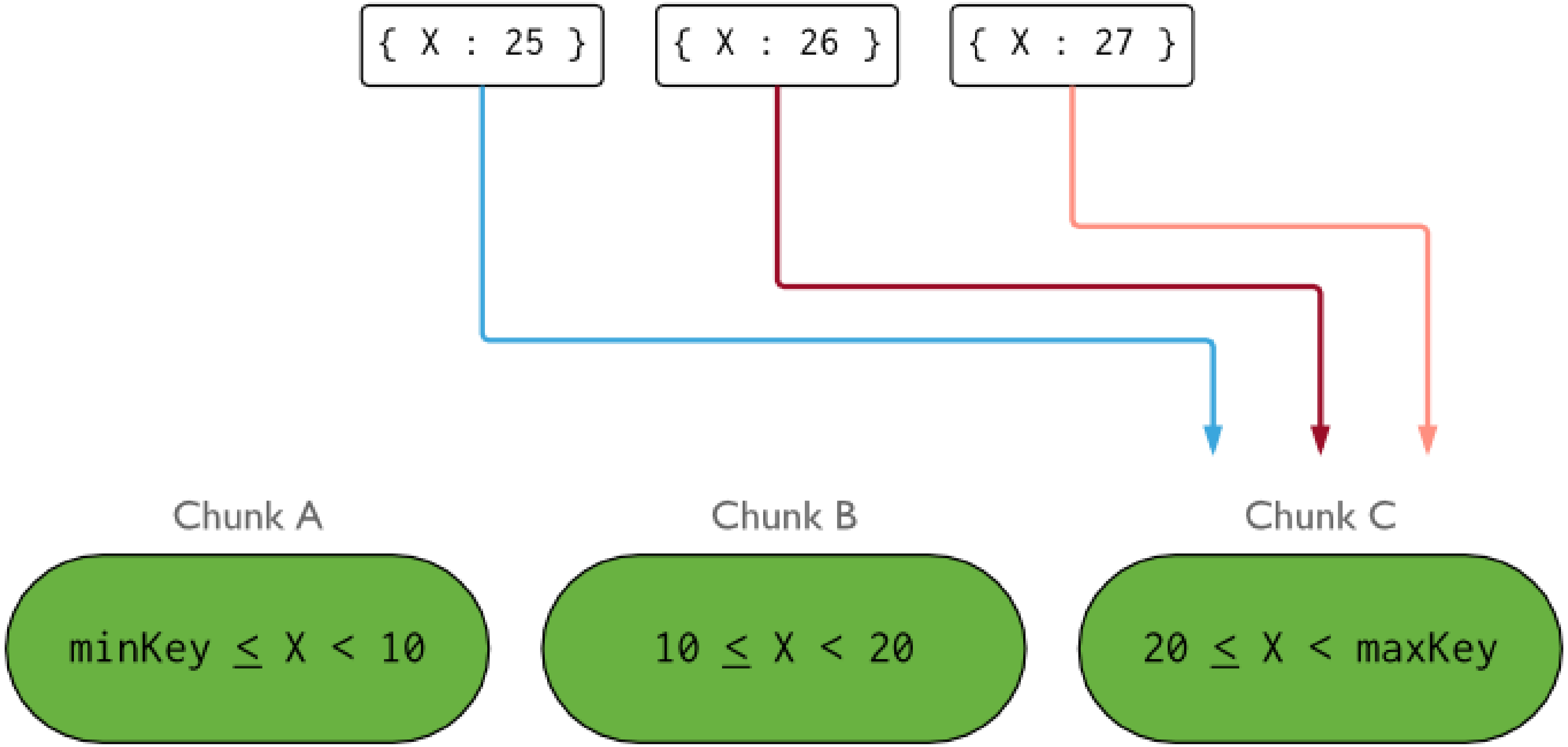
- The frequency of the shard key represents how often a given value occurs in the data.
- If the majority of documents contain only a subset of those values, then the chunks storing those documents become a bottleneck within the cluster.
- Furthermore, as those chunks grow, they may become indivisible chunks as they cannot be split any further.

Shard Key with High Frequency

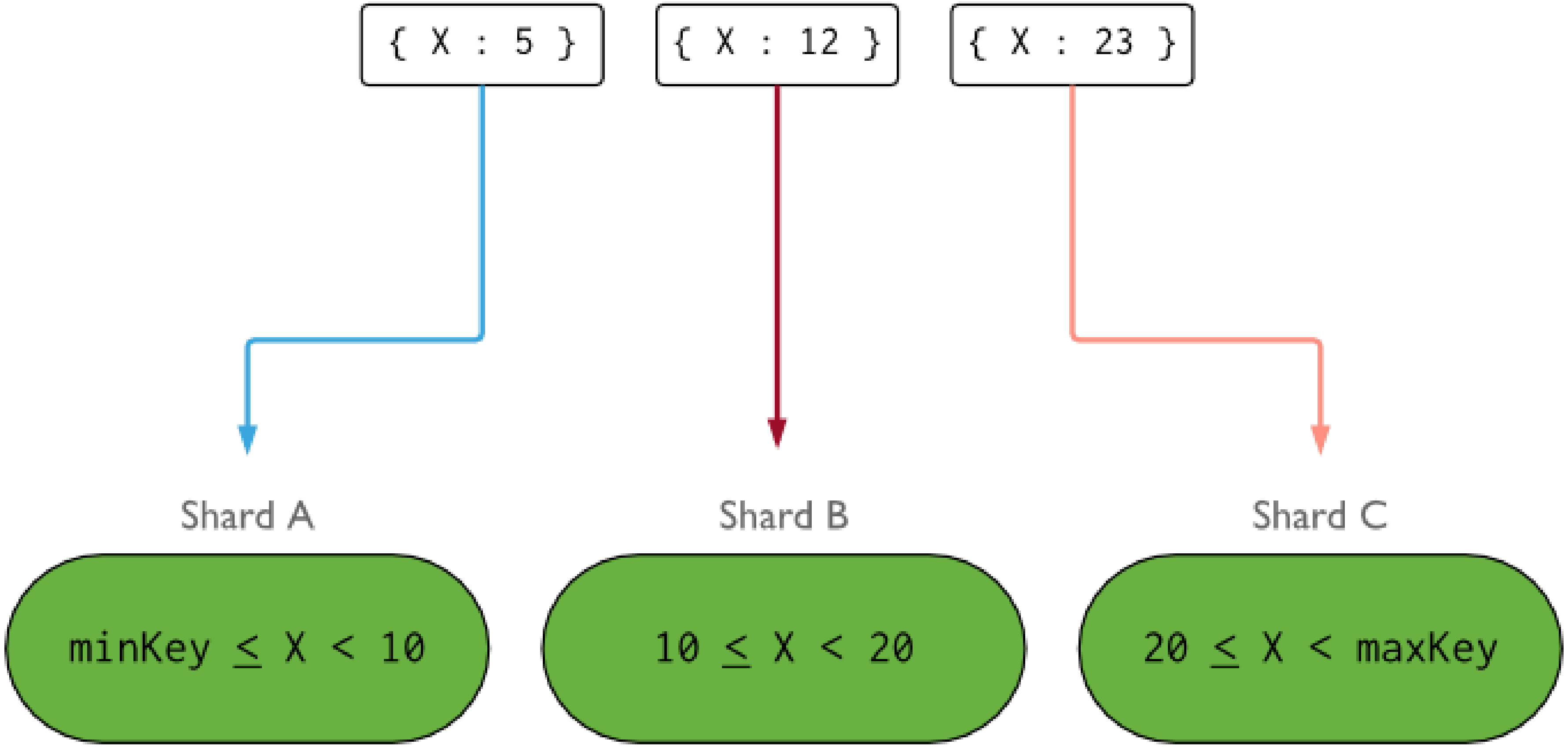


Ascending (or Descending) shard keys

The max chunk (the chunk with $maxKey) becomes bottleneck.

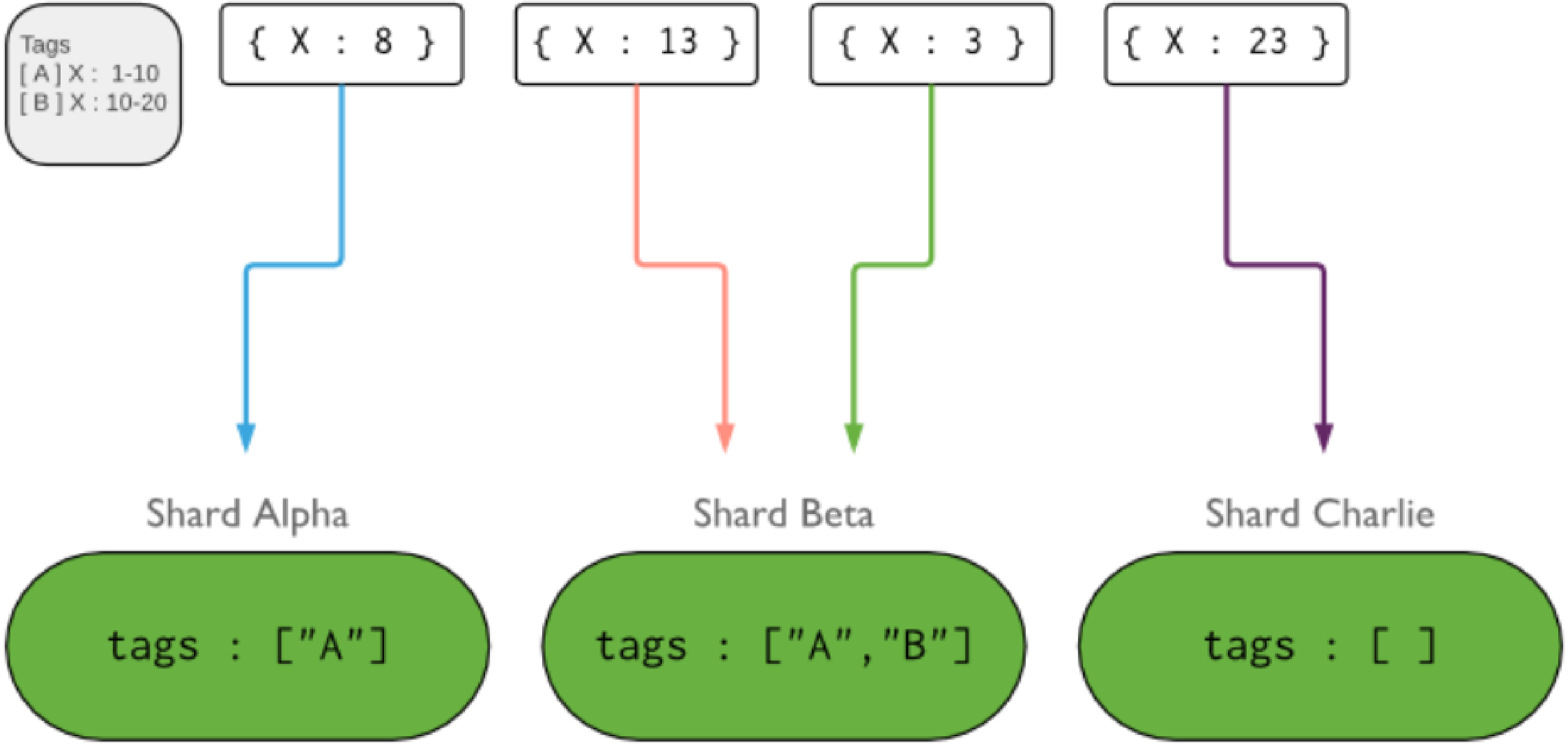


A shard cluster using a well chosen shard key



Tag aware sharding

Tag aware sharding *tags* specific ranges of the *shard* key and associates those *tags* with a *shard* or subset of shards.



Three shards and two tags. ;

Tag aware sharding

- The balancer respects tags and tag ranges during chunk migrations.
- For each chunk marked for migration, the balancer checks for destination shards with valid tag ranges.
- The balancer ensures tag ranges align with chunk boundaries by splitting chunks where necessary.
- Chunks that are not covered by a tag range can exist on *any* shard in the cluster and are migrated normally.
- During balancing rounds, if the balancer detects that any chunks violate the configured tags for a given shard, the balancer migrates those chunks to a shard where no conflict exists.

Tag aware sharding

1) Tag shards

```
mongos> sh.addShardTag("shard0000", "US");
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
mongos> sh.addShardTag("shard0001", "EU");
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
mongos> sh.status()
shards:
    {  "_id" : "shard0000",  "host" : "127.0.0.1:27023",  "tags" : [ "US" ] }
    {  "_id" : "shard0001",  "host" : "127.0.0.1:27024",  "tags" : [ "EU" ] }
```

2)Include the tag in the shard key (as the first element)

```
mongos> sh.shardCollection("testdb.testtagsharding", {region:1,
testkey:1})
{ "collectionsharded" : "testdb.testtagsharding", "ok" : 1 }
```

3) Adding a rule to each shard:

```
mongos>
sh.addTagRange("testdb.testtagsharding",
{region:"US"},{region:"MaxKey"},"US")
mongos>
sh.addTagRange("testdb.testtagsharding",
{region:MinKey},{region:"US"},"EU")
```

name space of the collection

minimum range (inclusive)

maximum range (exclusive)

tag

```
mongos> sh.status()
testdb.testtagsharding
                shard key: { "region" : 1, "testkey" : 1 }
                unique: false
                balancing: true
                chunks:
                        shard0000       1
                        shard0001       2
                { "region" : { "$minKey" : 1 }, "testkey" : { "$minKey" : 1 } } -->> { "region" :
"EU", "testkey" : { "$minKey" : 1 } } on : shard0001 Timestamp(2, 1)
                { "region" : "EU", "testkey" : { "$minKey" : 1 } } -->> { "region" : "US",
"testkey" : { "$minKey" : 1 } } on : shard0001 Timestamp(1, 3)
                { "region" : "US", "testkey" : { "$minKey" : 1 } } -->> { "region" : { "$maxKey"
: 1 }, "testkey" : { "$maxKey" : 1 } } on : shard0000 Timestamp(2, 0)
                 tag: EU  { "region" : { "$minKey" : 1 } } -->> { "region" : "US" }
                 tag: US  { "region" : "US" } -->> { "region" : "MaxKey" }
```

Populate data

```
mongos> for (i=0; i < 10000; i++)
{db.getSiblingDB("testdb").testtagsharding.insert({region:"EU",
testkey:i})}

WriteResult({ "nInserted" : 1 })
mongos>
mongos> db.testtagsharding.count()
10000
```

Shard 0000 tagged “US”

```
vagrant@vagrant-ubuntu-trusty-64:~$ sudo mongo 127.0.0.1:27023
> use testdb
switched to db testdb
> db.testtagsharding.count()
0
```

Shard 0001 tagged “EU”

```
vagrant@vagrant-ubuntu-trusty-64:~$ sudo mongo 127.0.0.1:27024
> use testdb
switched to db testdb
> db.testtagsharding.count()
10000
```

Replication vs. Sharding

- Replication creates an exact copy of data on multiple servers, so every server in a replica set is a mirror image of every other server.
- Every shard contains a different subset of data.