# MongoDB: Distributed NoSQL database

Martin Carrasco
*Computer Science*
*UTEC*
Lima, Peru
martin.carrasco@utec.edu.pe

Sebastian Carcamo
*Computer Science*
*UTEC*
Lima, Peru
sebastian.carcamo@utec.edu.pe

Jorge Ignacio Fiestas
*Computer Science*
*UTEC*
Lima, Peru
jorge.fiestas@utec.edu.pe

## I. Introduction

### A. Description & Motivation

*MongoDB* was originally developed as a component for a *platform as a service* product. These were web based application that looked to simplify the development and deployment process of web based app for developers. For this reason, MongoDB was conceived with two main purposes: 1) Being flexible and simple to use, as developers might need to use it for a wide range of applications; and 2) Being scalable in order to allow to work with hugh**mongo**us amounts of data (that is were it gets its name).

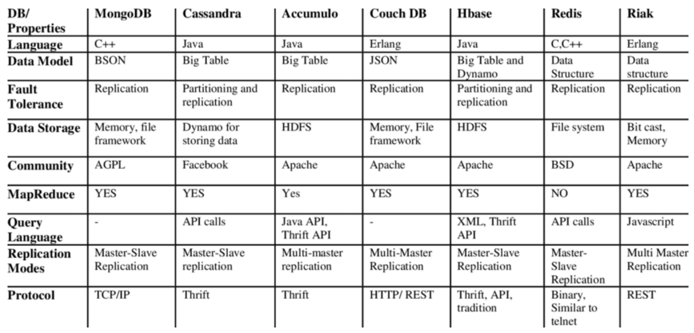
MongoDB is a document based non-relational database model that works with three main components: documents, fields and collections. *Documents* would be the MongoDB equivalent of a register in a traditional SQL database. Each documents has *fields*, that act like the attributes or columns of each register. This documents stored in BSON, which is a JSON file that is stored as a binary to optimize memory. When we group some documents we get a *collection*, that are the closest thing MongoDB has to tables and schemas.

One of the key differences between MongoDB and relational database models is that documents in the same collection do not need to share structure. That is to say that we can have documents with different fields and that not all objects with the same field have to store the same type of data. This also allows for the stored data to have more semantic meaning, and therefore, be more *human readable*.

### B. Comparison with other Non-Relational DB

MongoDB is currently established as the most popular non-relational database model in the market. This is probably due to three main differentiating factors:

1) As MongoDB was one of the few DB developed in `C++` it doesn't have the overhead that some of its competitors have, allowing it to be fast and lean.
2) MongoDB data storage is based on the filesystem of the native OS of the system in which it is working, this make it portable and lightweight, as it doesn't depend on external filesystems or further abstraction layers.
3) Even though it doesn't have its own query language, there exist multiple MongoDB libraries for almost all programming languages, specially web-oriented ones.



| DB/ Properties | MongoDB | Cassandra | Accumulo | Couch DB | Hbase | Redis | Riak |
|---|---|---|---|---|---|---|---|
| Language | C++ | Java | Java | Erlang | Java | C,C++ | Erlang |
| Data Model | BSON | Big Table | Big Table | JSON | Big Table and Dynamo | Data Structure | Data structure |
| Fault Tolerance | Replication | Partitioning and replication | Replication | Replication | Partitioning and replication | Replication | Replication |
| Data Storage | Memory, file framework | Dynamo for storing data | HDFS | Memory, File framework | HDFS | File system | Bit cast, Memory |
| Community | AGPL | Facebook | Apache | Apache | Apache | BSD | Apache |
| MapReduce | YES | YES | Yes | YES | YES | NO | YES |
| Query Language | - | API calls | Java API, Thrift API | - | XML, Thrift API | API calls | Javascript |
| Replication Modes | Master-Slave Replication | Master-Slave replication | Multi-master replication | Multi-Master Replication | Master-Slave Replication | Master-Slave Replication | Multi Master Replication |
| Protocol | TCP/IP | Thrift | Thrift | HTTP/ REST | Thrift, API, tradition | Binary, Similar to telnet | REST |

Fig. 1. A comparison between different non-SQL DB Models.

### C. Use Cases & Applications

MongoDB main use case is in the development of web based applications that need to evolve quickly and scale elegantly. This makes sense as it's JSON based file storage is highly compatible with most front-end programming languages. As well, thanks to it's simple file structure, MongoDB is good dealing with multidimensional data such as text, geospatial data and time series. As there is no restrictions to which documents get grouped in a collection, MongoDB is also perfect for the representation of data with natural clusters or high variability.

Thanks to its flexibility and reliability, MongoDB is used by tech leads businesses such as Adobe, Google, Nokia, SAP and many more.

## II. Characteristics description

In this section, we will talk about the functionalities that MongoDB has to offer. In this particular case, we will describe specifically about the MongoDB shell method instead of using APIs since this will give us more of an inside look of what's really happening in detail.

### A. Data model

First of all, its worth mentioning that MongoDB gives the developers a lot of control over something called *durability* in their write and read calls. Said durability is a property that guarantees that the written data will be saved and be permanent. This leads to a decision to be taken by the developer as a strong durability parameter will be safer to use since its guaranteed that the data will be saved on disk whilst a weak durability parameter will lead to a higher throughput but provide less data safety.

*1) Insertion:* The insertion method is used to insert one or more documents to a collection. Said documents need to be tagged with an id, whether its given by the user or not. If the user doesn't provide a unique *_id* field the daemon will assign it one.

```
db.products.insert( { item: "card",
 ↪  qty: 15 } )
```

```
{ "_id" :
 ↪  ObjectId("5063114bd386d8fadbd6b004"),
 ↪  "item" : "card", "qty" : 15}
```

Listing 1: Input and output of an insertion without specifying _id

```
db.products.insert( { _id: 10, item:
 ↪  "box", qty: 20 } )
```

```
{ "_id" : 10, "item" : "box", "qty" :
 ↪  20 }
```

Listing 2: Input and output of an insertion specifying _id

Multiple documents can be inserted if you pass an array of documents as the argument to the insert() method.

```
db.products.insert(
   [
     { _id: 11, item: "pencil", qty:
      ↪  50, type: "no.2" },
     { item: "pen", qty: 20 },
     { item: "eraser", qty: 25 }
   ]
)
```

```
{ "_id" : 11, "item" : "pencil", "qty"
 ↪  : 50, "type" : "no.2" }
{ "_id" :
 ↪  ObjectId("51e0373c6f35bd826f47e9a0"),
 ↪  "item" : "pen", "qty" : 20 }
{ "_id" :
 ↪  ObjectId("51e0373c6f35bd826f47e9a1"),
 ↪  "item" : "eraser", "qty" : 25 }
```

Listing 3: Input and output of an insertion of multiple documents

Regarding the starting introduction to durability, we have the *writeConcern* parameter. We will talk more in detail about this parameter later.

*Ordered* is another interesting parameter that MongoDB gives us choice to modify. When inserting in bulk, the default value for ordered is true. Doing an ordered insert implies that the insertions will be done serially and if an error occurs during one of those insertions, MongoDB will return without processing any of the remaining write operations in the list. On the other hand, an unordered insertion writes in parallel and will not stop writing if one of the insertions fails.

Lastly, its worth mentioning that the insertOne() and insertMany() methods are preferred since what they return gives more details about the insertions made.

*2) Deletion:* The deletion is done with the remove() method. This method takes the deletion criteria as a *query* parameter. It also has a parameter *justOne* that when set to true will only delete the first element that it finds that matches the query. Similarly to the insert method, the remove method also has a *writeConcern* parameter. The last parameter is *collation*, this parameter allows users to specify language-specific rules for string comparison.

Again, the alternative deleteOne() and deleteMany() methods are preferred because of their return values.

*3) Updates:* Updates can be performed in MongoDB collections. To do so, we make use of the updateOne() and updateMany() methods. Both methods work similarly. The first argument that they take is the condition that the document must meet to be modified. The second argument is the list of operations to be done to those documents.

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     arrayFilters: [ <filterdocument1>,
      ↪  ... ],
     hint:  <document|string>
   }
)
```

Listing 4: Update() method

A list with the update operators that can be applied to the documents can be found in the MongoDB documentation

*4) BulkWrite:* All the previous methods don't have to be written one by one, they can all be performed inside a bigger method called bulkWrite(). This method takes all the operations and uses the specified *writeConcern* and *ordered* parameters and uses it for all of them.

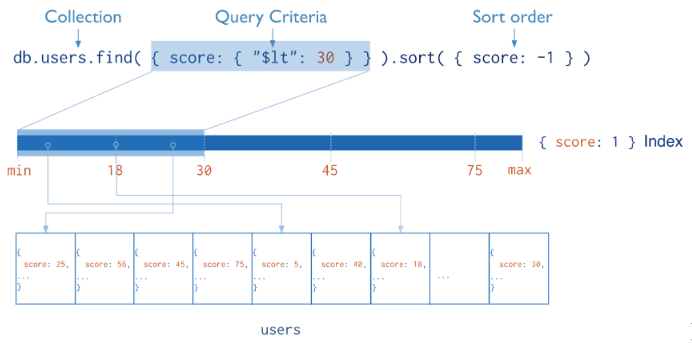


Fig. 2. Caption

```
db.collection.bulkWrite(
   [ <operation 1>, <operation 2>, ...
   ↪  ],
   {
      writeConcern : <document>,
      ordered : <boolean>
   }
)
```

Listing 5: bulkWrite() method

*5) Search:* To select documents in a collection or view we use the find() method. The first parameter that it receives is the query, where you can use any of these MongoDB query operators. The second parameter specifies the fields of the matching documents to be shown. Both of these parameters are optional and in case the user runs the method without parameters, the result will be all the documents in the collection.

*6) Indexing:* Indexes support the efficient execution of queries in MongoDB. Without them, a scan in every single document in the collection should be done to find the documents that match a query, which is very inefficient. Indexing limits the amount of documents MongoDB needs to inspect. MongoDB uses the B-tree data structure to allow the users to create indexes that store a small portion of the collection's data set in an easy to traverse form.

MongoDB creates a unique index on the _id field when creating a collection and also allows the users to create indexes based on their selected fields.

```
db.collection.createIndex( <key and
↪  index type specification>,
↪  <options> )
```

Listing 6: Command to create an index on a collection

## III. Distributed storage architecture

### A. Sharding

Given a data set too large for a single server to handle or high throughput applications we face a number of problems: high query rates exhaust the CPU capabilities of a server. Working set sizes larger than the server's RAM capacity stress the I/O capacity of disk drives.

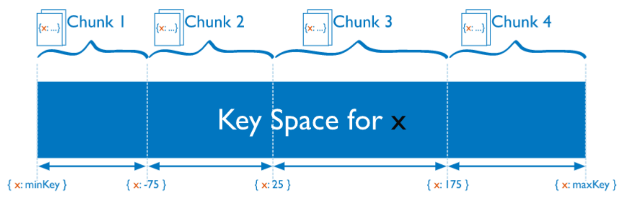


Fig. 3. Range-based sharding

To solve these problems we need to address the two types of scaling to handle system growth: vertical and horizontal scaling.

Vertical scaling consists in increasing the capabilities of a single server so its basically giving it more RAM or increasing the storage space. This type of scaling is limited by the current available technology.

Horizontal scaling consists in dividing the workload between multiple servers. The speed or capacity of a single server may not be high but while it works together with more servers, the load on a single server is diminished. Nonetheless, the trade-off for gaining efficiency is adding complexity to the infrastructure and maintenance to the deployment.

Adding this horizontal scaling in MongoDB is referred as *sharding*. To do this, we need to specify a shard key. A shard key is a field that must exist in every document, that will be used to distribute the documents across shards in chunks. Said chunks have inclusive lower and exclusive upper range based on the shard key.

Theres two types of sharding that can be used within MongoDB: Range-based sharding and Hash-based sharding. The range-base sharding divides the data sets into different ranges based on the value of the shard key. Documents having close shard key values are more likely to be in the same shard. This type of sharding enables us to do range queries based on the shard key values since its very probable that the adjacent document is located in the same shard. However, this partitioning can be uneven and will fail to scale if a small set of shards receive most of the requests.

On the other hand, the hash-based sharding is done by calculating the hash of the document's shard key value and then distributing the documents according to the result. This results in an evenly distributed data among the shards but doesn't guarantee that the documents with close shard key values reside in the same shard. This renders ineffective the range queries.

A MongoDB sharded cluster consists of the following components:

shards: contain subsets of the data set. mongos: Act as query router and provide an interface between client applications and the sharded cluster config servers: store metadata and configuration settings.

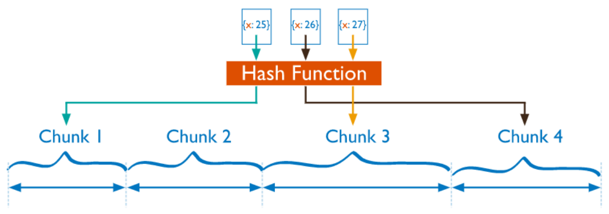


Fig. 4. Hash-based sharding

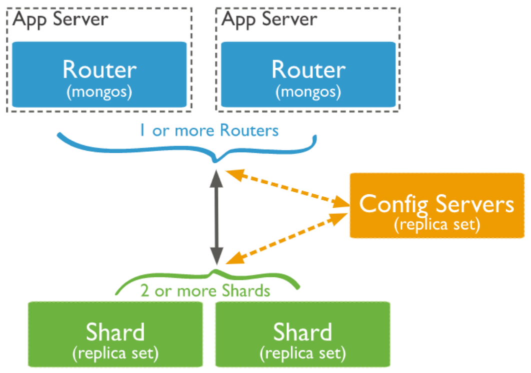


Fig. 5. Components

## B. Considerations before sharding

Sharding a data set requires infrastructure setup, which increases the deployment complexity. Consider sharding clusters when these apply:

- The data set is larger than a single MongoDB storage capacity.
- The size of the active working set exceeds the capacity of the maximum available RAM.
- A single MongoDB instance is unable to manage write operations.

Also, a pertinent choice of shard key is needed for ensuring efficiency and maximize performance as this key cannot be changed later.

## C. Replication

A replica set is a group of mongod processes that maintain the same data set. Having replica sets provides redundancy and high availability of the data when needed.

The sequence for writing data considering the replica sets is the following:

1) The write operations go to the primary node
2) The changes are recorded into the operation log
3) The write operations are replicated to the secondary nodes
4) Secondaries copy the primary oplog

The write operations on the secondary nodes and copying the primary oplog are done asynchronously. Having the replica set setup like this allows us to continue to function despite one or more nodes to be down or have failed.

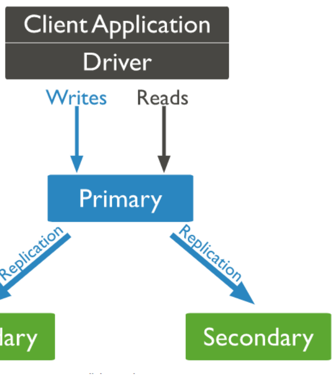


Fig. 6. Connections between nodes

When the primary node does not communicate with the secondaries after a certain amount of time, an election for a new primary node is done within all the nodes so a new primary is assigned and the functions can continue. The replica set cannot process write operations until the election completes successfully. The replica set can continue to serve read queries if such queries are configured to run on secondaries while the primary is offline.

The configuration options for the replica sets are:

- 50 members per replica set (7 voting members)
- Arbiter node:
  - Does not hold a copy of the data
  - Only used to vote in the elections to form a majority
- Priority 0 node:
  - Never becomes primary
  - Visible to the application
- Hidden node:
  - Never becomes primary
  - Not visible to the application
  - Used for reporting or for backups
- Delayed node:
  - Has a delay for a certain amount of seconds
  - Never becomes primary
  - (Optional) Hidden
  - Votes in elections
  - Mainly used for backups
- Write concern:
  - w - number of mongod instances (replica members) that acknowledged the write operation
  - j - number of mongod instances that acknowledged that the write operation has been written to their respective journal
  - wtimeout - time limit to prevent write operations
- Read preference:
  - primary (default) - reads only from the primary node
  - primaryPreferred - reads from the primary as a default but will read from the secondaries in case of failure.

- secondary - reads only from a secondary node
- secondaryPreferred - reads from the secondaries as a default but will read from the primary in case of failure.
- nearest - will read from the node that has the least network latency

## IV. Implementation

### A. Dataset information

We used the dataset provided as one of the examples from MongoDB. This is the *Amsterdam AirBnB* dataset. It contains arround 19K entries for AirBnB offers. Also taken from the MongoDB page here is an extract with a sample document general properties. The values of the fields are omitted to preserve the order in the document. We inserted this entries and indexed on the field *id* and *host_since*.

```
{
 "_id":,
 "listing_url":,
 "name": ,
 "summary": ,
 "interaction":,
 "house_rules": ,
 "property_type": ,
 "room_type": ,
 "bed_type": ,
 "minimum_nights":,
 "maximum_nights": ,
 "cancellation_policy": ,
 "last_scraped":,
 "calendar_last_scraped": ,
 "first_review": ,
 "last_review": ,
 "accommodates": ,
 "bedrooms": ,
 "beds": ,
 "number_of_reviews": ,
 "bathrooms": ,
 "amenities": ,
 "price": ,
 "security_deposit": ,
 "cleaning_fee": ,
 "extra_people": ,
 "guests_included": ,
 "images": ,
 "host": ,
 "address": ,
 "availability": ,
 "review_scores": ,
 "reviews":
}
```

Listing 1. Document sample

### B. Setup and configuration of MongoDB

*1) Basic setup:* The configuration we used is the following. We have 3 configuration servers, 3 shards with a replica each and a router. The *docker_compose.yml* used to deploy the services are at the Appendix aswell as the scripts for the configuration of each shard, the router and the configuration server. We will also reference the scripts used in our compose. Essentially we follow the tutorial referenced in the bibliography and sharded our collection according to some fields. To test the queries and load the dataset into MongoDB we decided to use Mongo Compass, a GUI that allows us to perform queries, upload data and, most importantly, analyze our queries with graphs to be able to better understand what is going on behind the curtains. Now we proceed to explain the steps we took next, which will be done in Mongo Compass, since all the configuration is already in place (except for the creation of the index and sharding) for us to start using MongoDB. In Figure 7 we can see the configuration using Mongo Compass about our sharding setup. We can also see in Figure 8 how our data is represented in Mongo Compass. We will later use some functions like filtering to run queries on our data and do the explain analyze of what happened with these queries.

*2) Index and collection sharding:* To test the performance of our databse we will run the same 3 queries in 2 setups. First we use a composite index of 3 fields *bedroom_type*, *room_type* and *property_type*. This one has no sharding. For the second setup we use a hash index on each of the fore mentioned fields and performed a sharding of the collection on *room_type*.

### C. Results and performance

To test our MongoDB setup we loaded the data and performed indexing as said before previously and the sharding. We will analyze the 2 setups. We will explain the queries and show the outputs of the explain analyze as follows.

*1) Setup 1: No sharding:*

- Here we filter all AirBnB listings that contain 1 bedroom and 2 beds and sort by the host_since date field and limit results at 1000. The total execution time is 114ms and the execution time performed in the shard01 is 54ms. We can see the results in Figure 9
- Here we perform and exact match to search for all the hosts named *Daniel* and we don't cap the results. We see here that the query takes 32ms and the time in the shard is 30ms. A reasonable time with 44 returned documents. We can see the results in 10

*2) Setup 2: Sharding on room_type:*

- Here we perform the same query as the first part, however we see that the total time is 204ms. This is due to gathering from all shards and then performing a merge. Since the query is not on the sharded attribute it is expected to be slower. 11
- We perform an exact match on a host name and the sharding actually increases the time from no sharding to 40ms but just a little. 12
- Finally we perform a query on one of the sharded attribute and sort by host since. This takes a time of of 100ms

which is way lower than the time it takes to run without sharding. This proves we need to be carefull about the indexing and sharding of these related to the type of work we want to do. 13

## V. Conclusion

After learning about how MongoDB performs in a distributed enviroment and how NoSQL data is handled by DBMS we can reach some final conclusions as to what is the value of MongoDB is the database ecosystem.

## References

[1] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22, 2013.

[2] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. Mongodb vs oracle–database comparison. In *2012 third international conference on emerging intelligent data and web technologies*, pages 330–335. IEEE, 2012.

[3] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[4] Peter Membrey, Eelco Plugge, and DUPTim Hawkins. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress, 2011.

[5] Thomas Stevens. mongo-shard-docker-compose. https://github.com/chefsplate/mongo-shard-docker-compose, 2018.

## Appendix

```
{
version: '2'
services:
  config01:
    image: mongo
    command: mongod --port 27017 --
       ↪ configsvr --replSet configserver
    volumes:
      - ./scripts:/scripts
  config02:
    image: mongo
    command: mongod --port 27017 --
       ↪ configsvr --replSet configserver
    volumes:
      - ./scripts:/scripts
  config03:
    image: mongo
    command: mongod --port 27017 --
       ↪ configsvr --replSet configserver
    volumes:
      - ./scripts:/scripts
  shard01a:
    image: mongo
    command: mongod --port 27018 --shardsvr
       ↪  --replSet shard01
    volumes:
      - ./scripts:/scripts
  shard01b:
    image: mongo
    command: mongod --port 27018 --shardsvr
       ↪  --replSet shard01
    volumes:
      - ./scripts:/scripts
  shard02a:
    image: mongo
    command: mongod --port 27019 --shardsvr
       ↪  --replSet shard02
    volumes:
      - ./scripts:/scripts
  shard02b:
    image: mongo
    command: mongod --port 27019 --shardsvr
       ↪  --replSet shard02
    volumes:
      - ./scripts:/scripts
  shard03a:
    image: mongo
    command: mongod --port 27020 --shardsvr
       ↪  --replSet shard03
    volumes:
      - ./scripts:/scripts
  shard03b:
    image: mongo
    command: mongod --port 27020 --shardsvr
       ↪  --replSet shard03
    volumes:
      - ./scripts:/scripts
  router:
    image: mongo
    command: mongos --port 27017 --configdb
       ↪  configserver/config01:27017,
       ↪ config02:27017,config03:27017 --
       ↪ bind_ip_all
    ports:
      - "27017:27017"
    volumes:
      - ./scripts:/scripts
    depends_on:
      - config01
      - config02
      - config03
      - shard01a
      - shard01b
      - shard02a
      - shard02b
      - shard03a
      - shard03b
```

Listing 2. docker_compose.yml

```
rs.initiate(
   {
     _id: "configserver",
     configsvr: true,
     version: 1,
     members: [
       { _id: 0, host : "config01:27017"
```

```
            ↪ },
        { _id: 1, host : "config02:27017"
            ↪ },
        { _id: 2, host : "config03:27017" }
      ]
    }
)
```

Listing 3. init-configserver.js

```
sh.addShard("shard01/shard01a:27018")
sh.addShard("shard01/shard01b:27018")

sh.addShard("shard02/shard02a:27019")
sh.addShard("shard02/shard02b:27019")

sh.addShard("shard03/shard03a:27020")
sh.addShard("shard03/shard03b:27020")
```

Listing 4. init-router.js

```
rs.initiate(
    {
      _id: "shard01",
      version: 1,
      members: [
        { _id: 0, host : "shard01a:27018"
            ↪ },
        { _id: 1, host : "shard01b:27018"
            ↪ },
      ]
    }
)
```

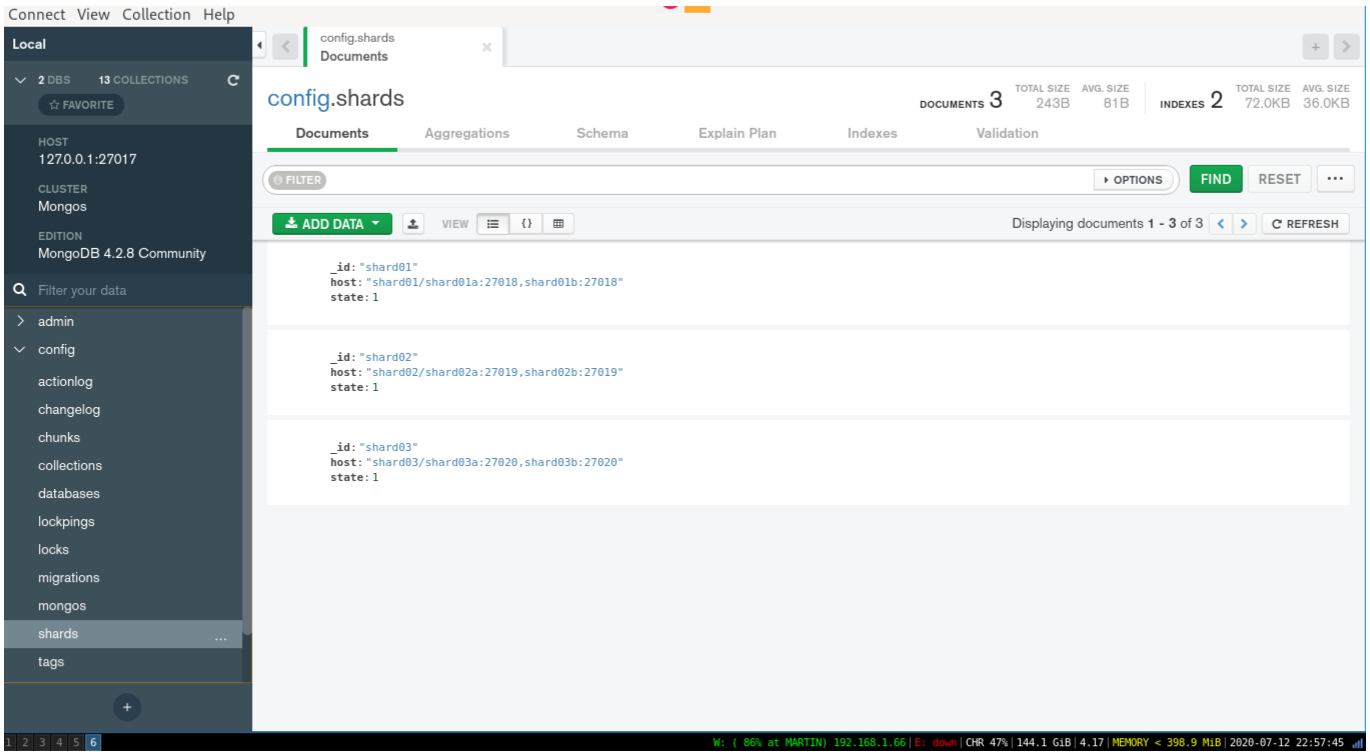
Listing 5. init-shard01.js

```
rs.initiate(
    {
      _id: "shard02",
      version: 1,
      members: [
        { _id: 0, host : "shard02a:27019"
            ↪ },
        { _id: 1, host : "shard02b:27019"
            ↪ },
      ]
    }
)
```

Listing 6. init-shard02.js

```
rs.initiate(
    {
      _id: "shard03",
      version: 1,
      members: [
        { _id: 0, host : "shard03a:27020"
            ↪ },
        { _id: 1, host : "shard03b:27020"
            ↪ },
      ]
    }
)
```

Listing 7. init-shard03.js



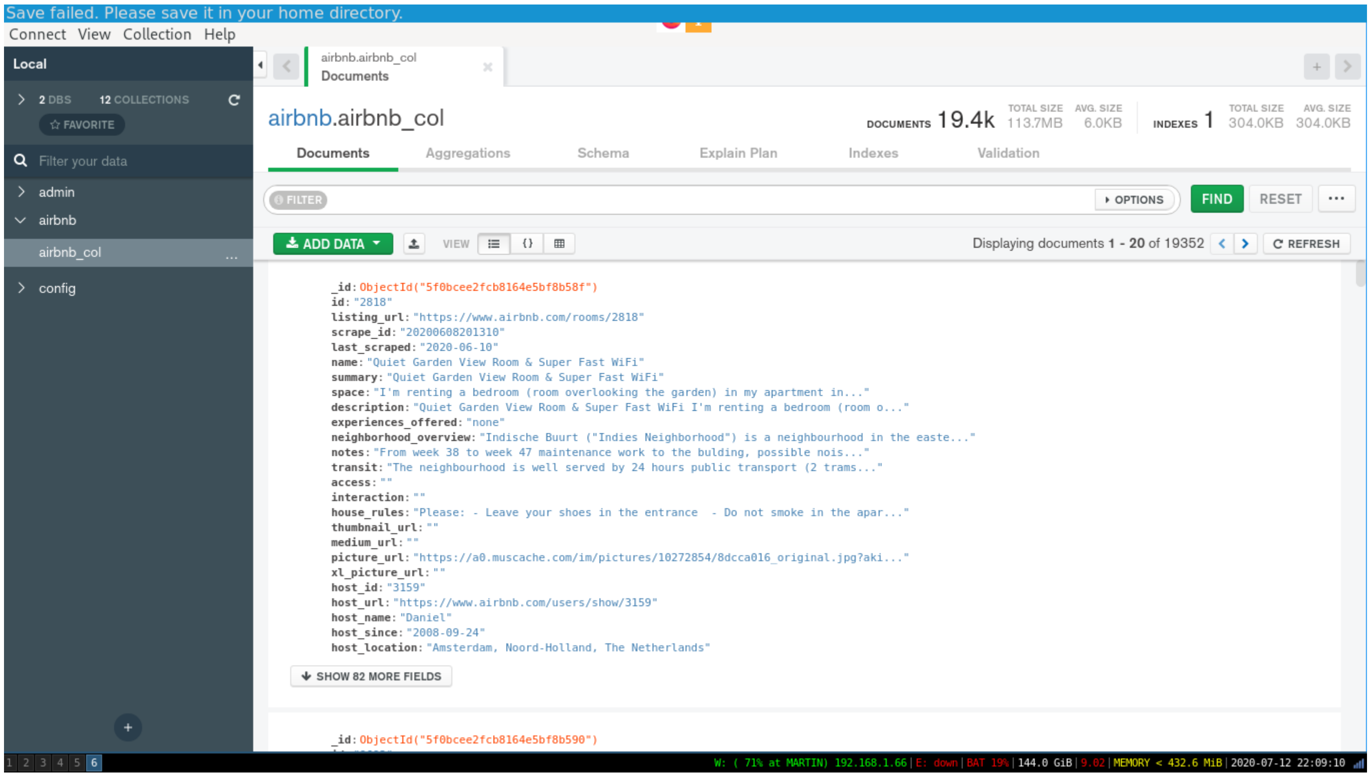
```
_id: "shard01"
host: "shard01/shard01a:27018,shard01b:27018"
state: 1
```

```
_id: "shard02"
host: "shard02/shard02a:27019,shard02b:27019"
state: 1
```

```
_id: "shard03"
host: "shard03/shard03a:27020,shard03b:27020"
state: 1
```

```
W: ( 86% at MARTIN) 192.168.1.66 | E: down | CHR 47% | 144.1 GiB | 4.17 | MEMORY < 398.9 MiB | 2020-07-12 22:57:45
```

Fig. 7. Sharding Setup

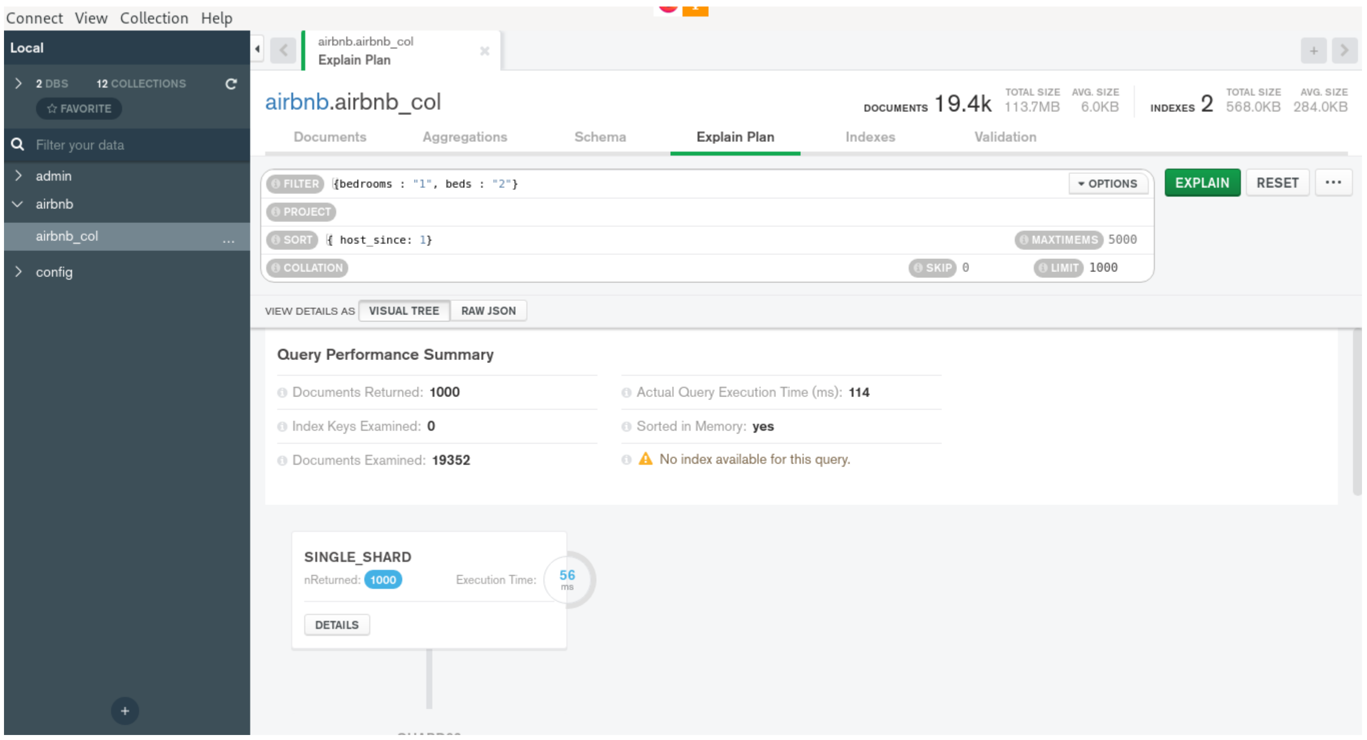


```
_id: ObjectId("5f0bcee2fcb8164e5bf8b58f")
id: "2818"
listing_url: "https://www.airbnb.com/rooms/2818"
scrape_id: "20200608201310"
last_scraped: "2020-06-10"
name: "Quiet Garden View Room & Super Fast WiFi"
summary: "Quiet Garden View Room & Super Fast WiFi"
space: "I'm renting a bedroom (room overlooking the garden) in my apartment in..."
description: "Quiet Garden View Room & Super Fast WiFi I'm renting a bedroom (room o..."
experiences_offered: "none"
neighborhood_overview: "Indische Buurt ("Indies Neighborhood") is a neighbourhood in the easte..."
notes: "From week 38 to week 47 maintenance work to the bulding, possible nois..."
transit: "The neighbourhood is well served by 24 hours public transport (2 trams..."
access: ""
interaction: ""
house_rules: "Please:  - Leave your shoes in the entrance  - Do not smoke in the apar..."
thumbnail_url: ""
medium_url: ""
picture_url: "https://a0.muscache.com/im/pictures/10272854/8dcca016_original.jpg?aki..."
xl_picture_url: ""
host_id: "3159"
host_url: "https://www.airbnb.com/users/show/3159"
host_name: "Daniel"
host_since: "2008-09-24"
host_location: "Amsterdam, Noord-Holland, The Netherlands"
```

```
_id: ObjectId("5f0bcee2fcb8164e5bf8b590")
```

```
W: ( 71% at MARTIN) 192.168.1.66 | E: down | BAT 19% | 144.0 GiB | 9.02 | MEMORY < 432.6 MiB | 2020-07-12 22:09:10
```

Fig. 8. Mongo Compass General View



airbnb.airbnb_col

DOCUMENTS 19.4k TOTAL SIZE 113.7MB AVG. SIZE 6.0KB INDEXES 2 TOTAL SIZE 568.0KB AVG. SIZE 284.0KB

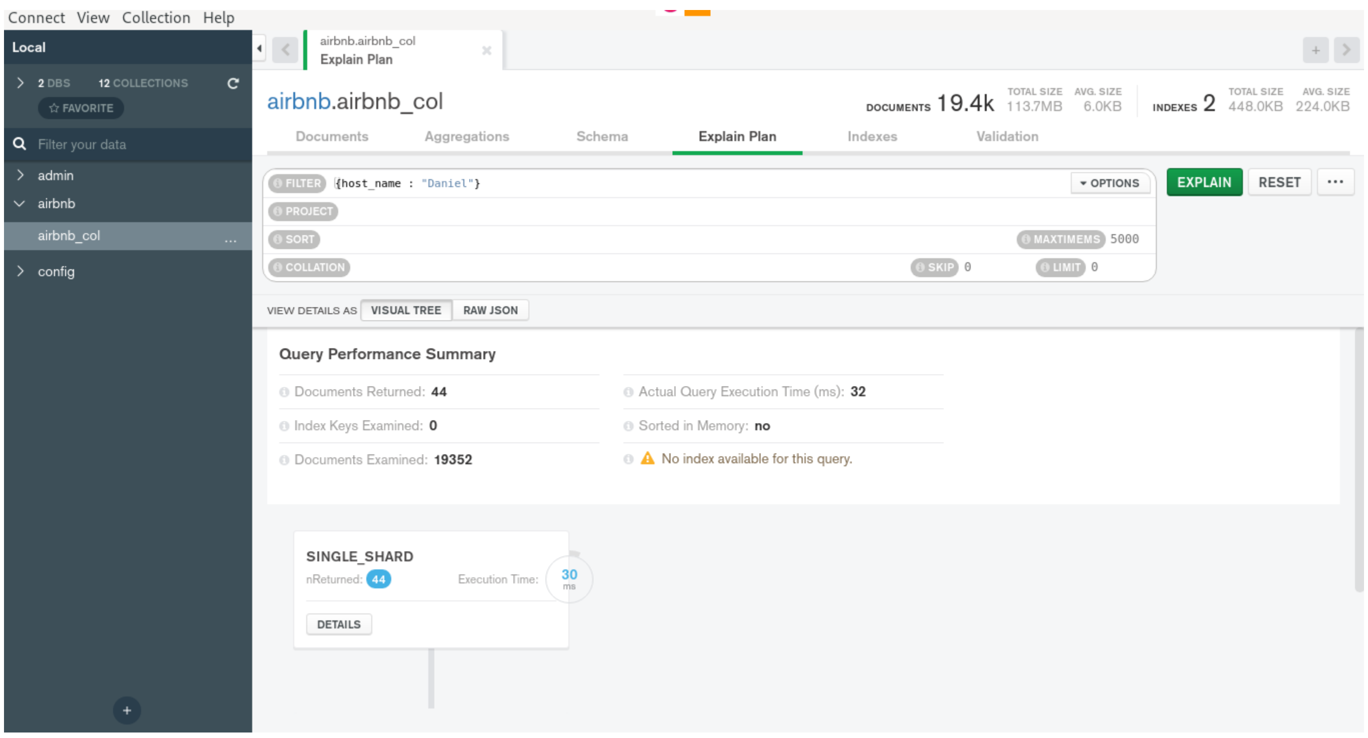
FILTER {bedrooms : "1", beds : "2"}
PROJECT
SORT { host_since: 1}
MAXTIMEMS 5000
COLLATION
SKIP 0
LIMIT 1000

Query Performance Summary

| | |
|---|---|
| Documents Returned: **1000** | Actual Query Execution Time (ms): **114** |
| Index Keys Examined: **0** | Sorted in Memory: **yes** |
| Documents Examined: **19352** | ⚠ No index available for this query. |

SINGLE_SHARD
nReturned: 1000
Execution Time: 56 ms

Fig. 9. Query 1



airbnb.airbnb_col

DOCUMENTS 19.4k TOTAL SIZE 113.7MB AVG. SIZE 6.0KB INDEXES 2 TOTAL SIZE 448.0KB AVG. SIZE 224.0KB

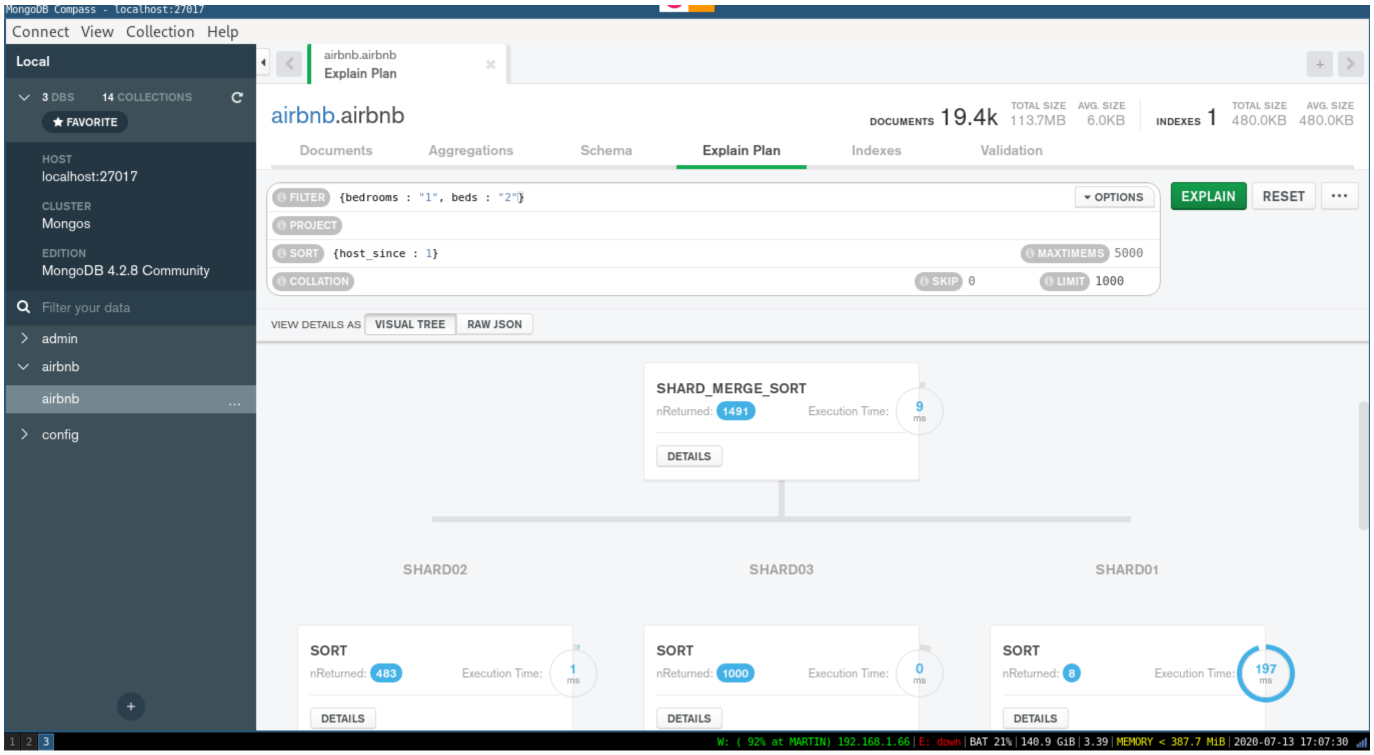
FILTER {host_name : "Daniel"}
PROJECT
SORT
MAXTIMEMS 5000
COLLATION
SKIP 0
LIMIT 0

Query Performance Summary

| | |
|---|---|
| Documents Returned: **44** | Actual Query Execution Time (ms): **32** |
| Index Keys Examined: **0** | Sorted in Memory: **no** |
| Documents Examined: **19352** | ⚠ No index available for this query. |

SINGLE_SHARD
nReturned: 44
Execution Time: 30 ms

Fig. 10. Query 2

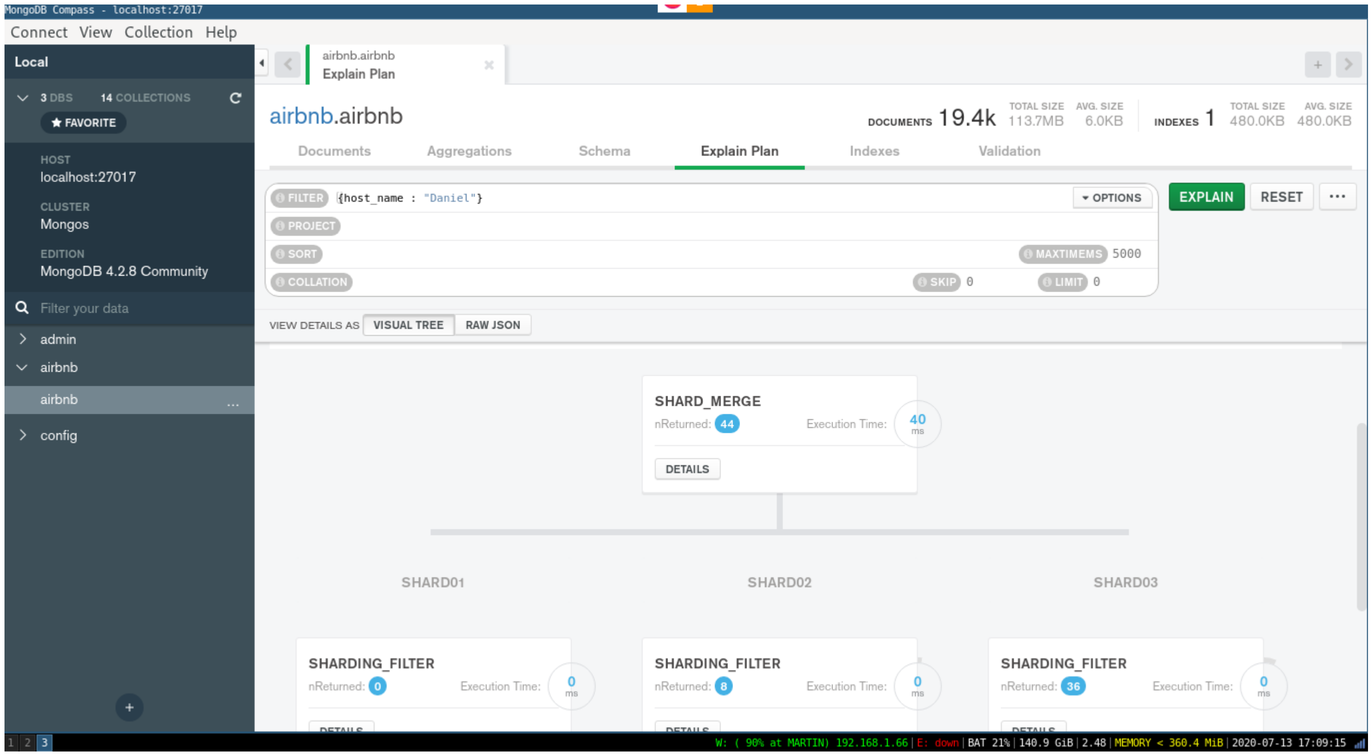

MongoDB Compass - localhost:27017
Connect View Collection Help
Local
3 DBS 14 COLLECTIONS
FAVORITE
HOST
localhost:27017
CLUSTER
Mongos
EDITION
MongoDB 4.2.8 Community
Filter your data
admin
airbnb
airbnb
config
airbnb.airbnb
Explain Plan
airbnb.airbnb
DOCUMENTS 19.4k TOTAL SIZE 113.7MB AVG. SIZE 6.0KB
INDEXES 1 TOTAL SIZE 480.0KB AVG. SIZE 480.0KB
Documents Aggregations Schema Explain Plan Indexes Validation
OPTIONS EXPLAIN RESET
VIEW DETAILS AS VISUAL TREE RAW JSON
SHARD02 SHARD03 SHARD01
DETAILS

```
FILTER     {bedrooms : "1", beds : "2"}
PROJECT
SORT       {host_since : 1}                         MAXTIMEMS 5000
COLLATION                               SKIP 0      LIMIT 1000

SHARD_MERGE_SORT   nReturned: 1491   Execution Time: 9 ms
SHARD02  SORT      nReturned: 483    Execution Time: 1 ms
SHARD03  SORT      nReturned: 1000   Execution Time: 0 ms
SHARD01  SORT      nReturned: 8      Execution Time: 197 ms

W: ( 92% at MARTIN) 192.168.1.66 | E: down | BAT 21% | 140.9 GiB | 3.39 | MEMORY < 387.7 MiB | 2020-07-13 17:07:30
```

Fig. 11. Query Sharded 1

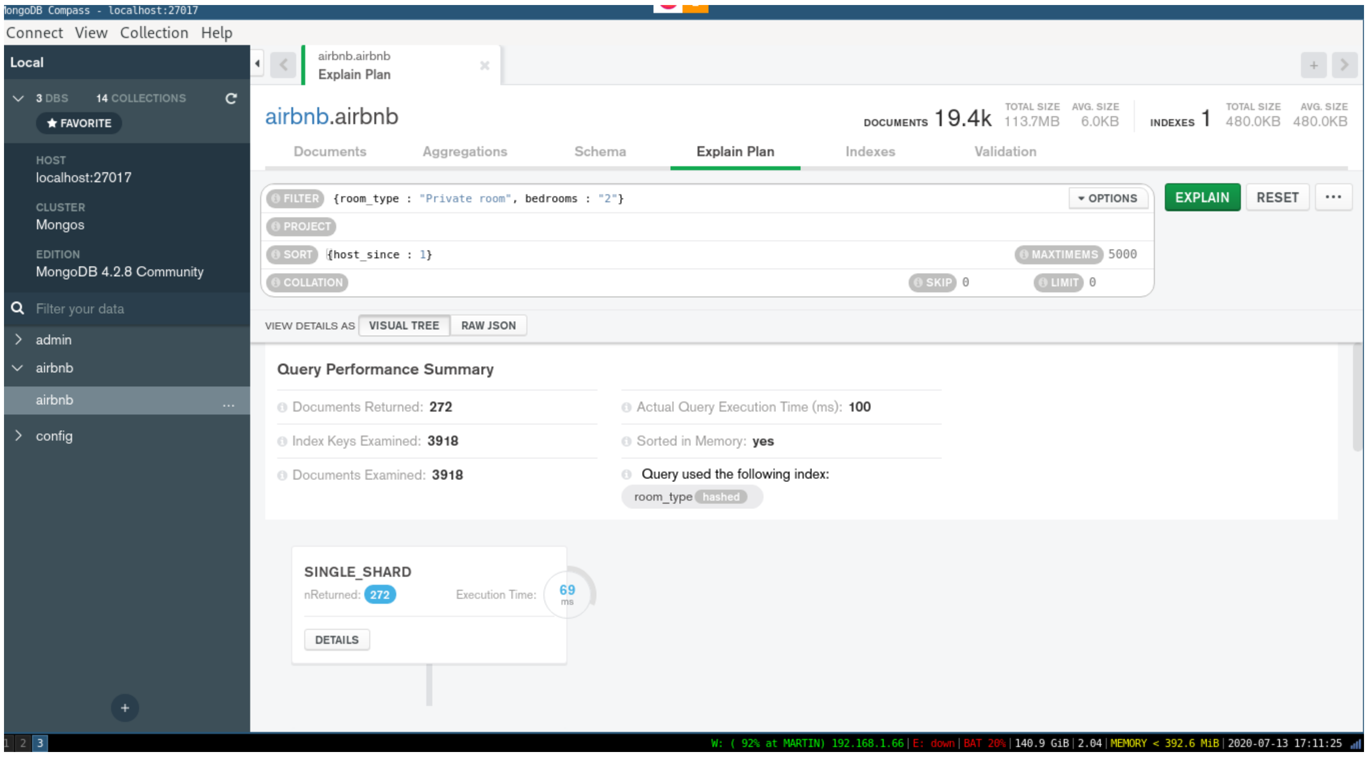

MongoDB Compass - localhost:27017
Connect View Collection Help
Local
3 DBS 14 COLLECTIONS
FAVORITE
HOST
localhost:27017
CLUSTER
Mongos
EDITION
MongoDB 4.2.8 Community
Filter your data
admin
airbnb
airbnb
config
airbnb.airbnb
Explain Plan
airbnb.airbnb
DOCUMENTS 19.4k TOTAL SIZE 113.7MB AVG. SIZE 6.0KB
INDEXES 1 TOTAL SIZE 480.0KB AVG. SIZE 480.0KB
Documents Aggregations Schema Explain Plan Indexes Validation
OPTIONS EXPLAIN RESET
VIEW DETAILS AS VISUAL TREE RAW JSON
SHARD01 SHARD02 SHARD03
DETAILS

```
FILTER     {host_name : "Daniel"}
PROJECT
SORT                                                MAXTIMEMS 5000
COLLATION                               SKIP 0      LIMIT 0

SHARD_MERGE                 nReturned: 44   Execution Time: 40 ms
SHARD01  SHARDING_FILTER    nReturned: 0    Execution Time: 0 ms
SHARD02  SHARDING_FILTER    nReturned: 8    Execution Time: 0 ms
SHARD03  SHARDING_FILTER    nReturned: 36   Execution Time: 0 ms

W: ( 90% at MARTIN) 192.168.1.66 | E: down | BAT 21% | 140.9 GiB | 2.48 | MEMORY < 360.4 MiB | 2020-07-13 17:09:15
```

Fig. 12. Query Sharded 2



HOST localhost:27017

CLUSTER Mongos

EDITION MongoDB 4.2.8 Community

airbnb.airbnb

DOCUMENTS 19.4k TOTAL SIZE 113.7MB AVG. SIZE 6.0KB

INDEXES 1 TOTAL SIZE 480.0KB AVG. SIZE 480.0KB

```
FILTER     {room_type : "Private room", bedrooms : "2"}
PROJECT
SORT       {host_since : 1}                         MAXTIMEMS 5000
COLLATION                                 SKIP 0    LIMIT 0
```

Query Performance Summary

| | |
|---|---|
| Documents Returned: **272** | Actual Query Execution Time (ms): **100** |
| Index Keys Examined: **3918** | Sorted in Memory: **yes** |
| Documents Examined: **3918** | Query used the following index: room_type hashed |

SINGLE_SHARD

nRetruned: 272 Execution Time: 69 ms

```
W: ( 92% at MARTIN) 192.168.1.66 | E: down | BAT 20% | 140.9 GiB | 2.04 | MEMORY < 392.6 MiB | 2020-07-13 17:11:25
```

Fig. 13. Query Sharded 3