**Chapter 4 – Scaling Experiments**

**Epic Games Store**

**CDOF3 - LE QUERNEC Loévan / MARTIN Santiago / RINGEISSEN Léo / SALEH Julien**

Table of Contents

[Performance measurement 2](#_Toc185370264)

[Python script 2](#_Toc185370265)

[Shards setup 3](#_Toc185370266)

[The results for 2, 4 and 6 shards 8](#_Toc185370267)

[Why these results? 9](#_Toc185370268)

[Administrator statistics 10](#_Toc185370269)

[2 shards 10](#_Toc185370270)

[4 shards 11](#_Toc185370271)

[6 shards 12](#_Toc185370272)

[Conclusion 13](#_Toc185370273)

The entire codes are available here: <https://github.com/Blxucreep/data-structures-for-cloud-project>.

Launch the VMs:



# Performance measurement

To measure the performance of the queries, we needed to:

* Adapt them to the fact that we can’t use *lookup* function because of the version of MongoDB (the denormalizations are from now on useful!).
* Developp a script to measure the performances.
* Discuss the results we have.

## Python script

Here’s a quick screen of the Python code, using *pymongo* to connect to the database:

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

The steps resumed:

* First, we connect to the database. We create variables for the identifiers of the connection, and then we need to use the MongoClient function connect to mongo from a local machine to one of the VMs.
* All the queries are regrouped in a single file, where they are all named with the function they use and the query itself. A function is defined for loading the queries.
* Then we have a function *test\_queries* to make the query.
* When the function *measure\_performance* is called at the same time to measure the time that the query makes to be executed.
* Finally, we print the results.

## Shards setup

Before reporting our results, we need to set up the shards by launching the other VMs and test the performances with 2, 4 and 6 shards, 1 (for us) has no interest. Here is the maximum configuration with 6 shards:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Server name | Role | Shard | ReplicaSet | Option |
| MESIIN59202401010 | mongos |  |  |  |
| MESIIN59202401009 | configSvr |  | config | arbiter:RS1,RS2,RS3,RS4,RS5,RS6 |
| MESIIN59202400046  MESIIN59202401011 | shard  shard | S1  S1 | RS1  RS1 | priority : 1  priority : 0.5 |
| MESIIN59202400047  MESIIN59202401012 | shard  shard | S2  S2 | RS2  RS2 | priority : 1  priority : 0.5 |
| MESIIN59202401011  MESIIN59202400020 | shard  shard | S3  S3 | RS3  RS3 | priority : 1  priority : 0.5 |
| MESIIN59202401012  MESIIN59202400021 | shard  shard | S4  S4 | RS4  RS4 | priority : 1  priority : 0.5 |
| MESIIN59202400020  MESIIN59202400046 | shard  shard | S5  S5 | RS5  RS5 | priority : 1  priority : 0.5 |
| MESIIN59202400021  MESIIN59202400047 | shard  shard | S6  S6 | RS6  RS6 | priority : 1  priority : 0.5 |

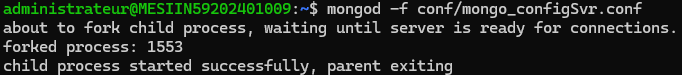
What we did:

* Create the replica set files for each VM (RS3, RS4, etc).
* Be sure that MongoDB is down (the one by default).

**sudo mongod -f /etc/mongodb.conf --shutdown**

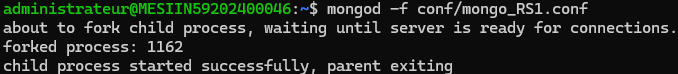
* Launch each server with its configuration file:
  + mongo\_configSvr for the configuration server,

**mongod -f conf/mongo\_configSvr.conf**

****

* + mongo\_RSX.conf for the shards,

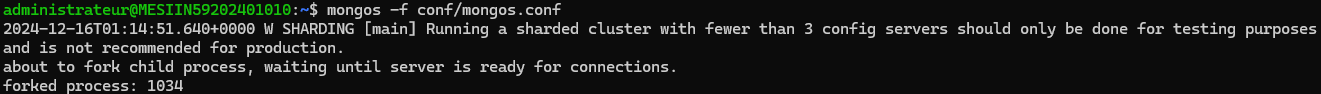
**mongod -f conf/mongo\_RSX.conf**

****

*(and same for the other shards)*

* + mongos.conf for the router (without forgetting configdb=configSvr/MESIIN59202401009:27018).

**mongos -f conf/mongos.conf**

****

* Launch the mongo shell:
  + **mongo --host XXX --port 27018** for the config server,
  + **mongo --host XXX --port 2701X** for the shards,
  + **mongo --host XXX --port 30000** for the router.
* Initiate the replica sets (except for the router).

**rs.initiate();**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

*(and same for all the shards)*

* Add the shards on the router (for the primary RS).

**sh.addShard("RSX/XXX:27017")**

**Une image contenant texte, capture d’écran, Police, noir

Description générée automatiquement**

*(and same for the other primaries)*

* And then the secondaries on the primaries.

**rs.add("XXX:27018")**

* Enable the sharding.

**sh.enableSharding("epic");**

**Une image contenant texte, capture d’écran, Police, noir

Description générée automatiquement**

* Create the collections (user and analyst).

**db.createCollection("games\_XXX\_view");**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

* Create the shard collection.

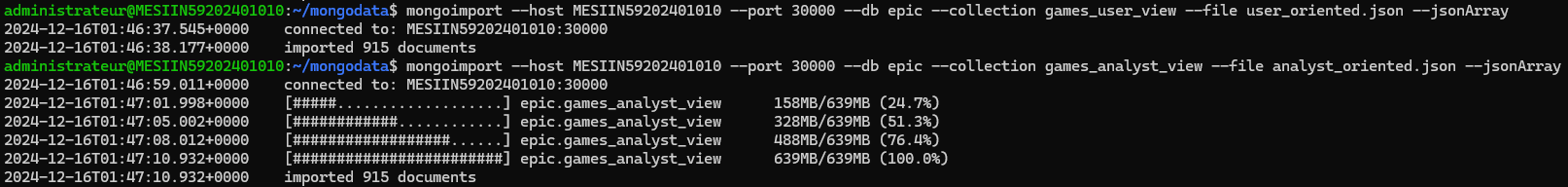
**sh.shardCollection("epic.games\_XXX\_view", {"\_id": "hashed"});**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

* Import the documents (not int the mongo shell).

**mongoimport --db epic --collection games\_XXX\_view --file XXX\_oriented.json –jsonArray**

****

All explained here is simply some screenshots we took to show some examples of how we did this. We did not captured everything, since we do the same commands for all the VMs.

* Verify the status.

**sh.status();**

Here, we see with our first test that everthing is working properly, our 2 first shards (RS1 and RS2) are listed:

Une image contenant texte, capture d’écran

Description générée automatiquement

## The results for 2, 4 and 6 shards

What we did is that we incremented the number of shards progressively. The first test was composed of 2 shards, and for each of them a primary and a secundary. After that, same goes for 4 and 6 shards. Here are our results, 2 times for each query (user and analyst denormalizations) and for each configuration:

|  |  |  |  |
| --- | --- | --- | --- |
| Query | 2 shards | 4 shards | 6 shards |
| 1 user | 0.032942 | 0.029702 | 0.028718 |
| 1 analyst | 0.032964 | 0.032970 | 0.029502 |
| 2 user | 0.020559 | 0.023040 | 0.021571 |
| 2 analyst | 0.022400 | 0.021608 | 0.022717 |
| 3 user | 0.020008 | 0.025141 | 0.023173 |
| 3 analyst | 0.024534 | 0.022049 | 0.022697 |
| 4 user | 0.020806 | 0.023222 | 0.022774 |
| 4 analyst | 0.021003 | 0.021304 | 0.020809 |
| 5 user | 0.032726 | 0.036516 | 0.037223 |
| 5 analyst | 0.033577 | 0.032428 | 0.032308 |
| 6 user | 0.085883 | 0.068542 | 0.060113 |
| 6 analyst | 0.579877 | 0.578418 | 0.470631 |
| 7 user | 0.055814 | 0.035557 | 0.040727 |
| 7 analyst | 0.054722 | 0.040668 | 0.036310 |
| 8 user | 0.111428 | 0.078022 | 0.052545 |
| 8 analyst | 0.072712 | 0.069581 | 0.060223 |

Here’s a screenshot of the results that are returned by the script:

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

With that, we executed the script 16 times (2 times per query, for each of the 2 denormalizations), for each sharding experience. The total number of executions is therefore 48.

## Why these results?

Increasing the number of shards significantly boosts performance for both User-oriented and Analyst-oriented queries. For User-oriented queries, execution times consistently decrease as shard counts increase from 2 to 6. This is due to the simpler query structure, as critical information like critics, social networks, and hardware configurations are embedded directly in game documents, reducing the need for joins. The increased number of shards allows for better distribution of queries, enabling parallel processing, which results in faster query execution.

For Analyst-oriented queries, a similar pattern is observed, though the performance gains are less significant. This is due to the larger document size and more complex queries, which often require aggregations and filtering. While sharding still reduces the load on individual servers, the processing of complex queries across multiple shards provides smaller but still noticeable improvements in execution time.

Overall, increasing the number of shards allows for more efficient parallel execution of queries. User-oriented queries benefit the most, as their structure requires less computation. Analyst-oriented queries, while more complex, still see performance gains from distributed processing. The results demonstrate how the choice of denormalization strategy and shard configuration directly impacts system performance.

# Administrator statistics

We divided this part into 3 subparts, each of them representing a configuration we tested (2, 4 and 6 shards). Let’s review the statistics!

## 2 shards

The sharding configuration for both collections:

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

*(forget to do it on the other collection here, sorry)*

The statisctics described here are useful for analysis purposes. We see the estimated data and docs per chunks, and exactly what is distributed for each shard.

## 4 shards

The screens show the sharding configuration for both of the user-oriented and analyst-oriented collections.Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, noir et blanc

Description générée automatiquement

## 6 shards

Our 6 shards are up, and well implemented:

Une image contenant texte, capture d’écran, noir, Police

Description générée automatiquement

For the collection “games\_analyst\_view”, everything is distributed through all the shards, but for the other one, “games\_user\_view”, thers’re only 4 chunks, then the data is distrubuted through 4 shards only (RS1 to RS4). We don’t know why the size of the chunks has not changed, but the result is that the times for the queries have not significally changed.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

# Conclusion

Through our scaling experiments with 2, 4, and 6 shards, we observed a general improvement in query performance as the number of shards increased, particularly for more complex queries. However, the results showed diminishing returns when scaling from 4 to 6 shards, suggesting that the benefits of sharding depend on how evenly data is distributed across chunks and shards. For instance, certain collections like "games\_user\_view" did not utilize all shards, limiting performance gains. Overall, our analysis highlights the importance of efficient chunk distribution and workload balancing when implementing a sharded MongoDB architecture.

Thanks for reading us!