Distributed Database Systems - Final Report

Armando Fortes[1], **David Pissarra**[2]
Distributed Database Systems Course Fall 2021
Department of Computer Science and Technology
Tsinghua University, Beijing, China
`{ferreiracardos10`[1]`, pissarrad10`[2]`}@mails.tsinghua.edu.cn`

Abstract

Distributed Database Systems have become the dominant data management tool for Big Data. Since it is the main purpose of this course project, we have built a Distributed Database System using MongoDB to store structured data and Hadoop Distributed File System for unstructured data. Furthermore, we developed a simple TKinter application, in order to combine everything together in a single interactive component, and a data cache using Redis, as similar requests to the database may happen. Every component of our architecture is running on a different Docker container, with the intention of simulating a distributed environment.
Source Code: `https://github.com/davidpissarra/ddbs-project`

1 Problem Background and Motivation

Big data can be described in terms of data management challenges which cannot be solved with traditional databases, due to increasing volume, velocity and variety of data [1]:

- **Volume:** Data is collected by organizations from a variety of sources, including videos, images, audio, social media, transactions, industrial equipment and more. Ranges from terabytes to petabytes of data [2].
- **Velocity:** Nowadays, data streams into businesses at an unprecedented speed and has to be handled in a timely manner. Therefore, data needs to be collected, stored, processed, and analyzed within relatively short windows. [1] [2]
- **Variety:** Data comes in all types of formats and from a wide range of sources (web logs, social media interactions, e-commerce and online transactions, etc.) [1]

The purpose of this project is to help the students gain a deep understanding of the advanced big data management techniques in a distributed environment through a hands-on software design and implementation experiment. Moreover, we will also be able to grasp the very latest big data technologies and apply them to solve a real-world problem, while nurturing the team-work spirit through cooperative work.

2 Existing Solutions

2.1 MongoDB Sharding

By using the MongoDB Sharding [3], it is possible to distribute data across multiple machines and geographic locations. Since we want to fragment our data, based on different attributes, shard keys can easily identify where chunks of data should be located. Sharding allows creating different replica sets, which will provide high availability over more than one MongoDB server. A MongoDB sharded cluster consists of the following components:

- **Shards:** Each shard contains a subset of the sharded data, and will be part of a replica set.
- **Config servers:** Store metadata and configuration settings for the sharded data.
- **Query router:** Provides an interface between client applications and the sharded cluster.

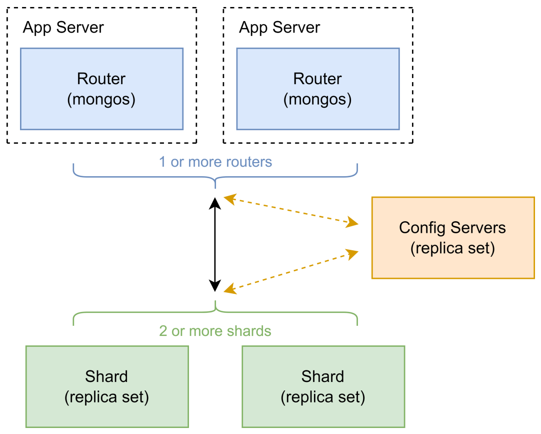


Figure 1: MongoDB Sharding Architecture.

2.2 MySQL Cluster

MySQL Cluster [4] also provides a distributed relational database with linear scalability and high availability. Even with a large transaction throughput the cluster can manage an excellent performance. In terms of architecture and data distribution both MongoDB and MySQL approaches will do it similarly. The MySQL Cluster Manager will be responsible of tuning every cluster and delivering query results with the highest availability.

2.3 Hadoop Distributed File System

HDFS (Hadoop Distributed File System) [5] is a highly fault-tolerant distributed file system providing high throughput access to application data and is suitable for applications that have large datasets, such as unstructured data. HDFS has a master/slave architecture composed by:

- **NameNode:** A master server that manages the file system namespace and regulates the access to files by clients.
- **DataNodes:** Nodes in the cluster which manage storage attached to the hardware where they run on.

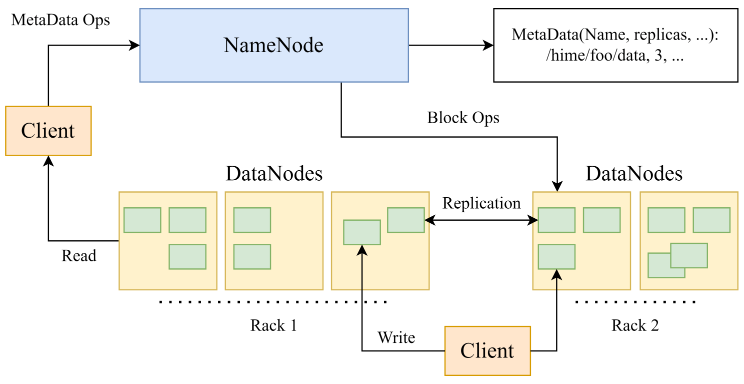


Figure 2: HDFS Architecture.

3 Problem Definition

Build a distributed data center, where the data to be managed and processed includes structured data (5 relational tables), whose inter-relations may be observed in Figure 3, and unstructured data (text, images and video), related to article contents.

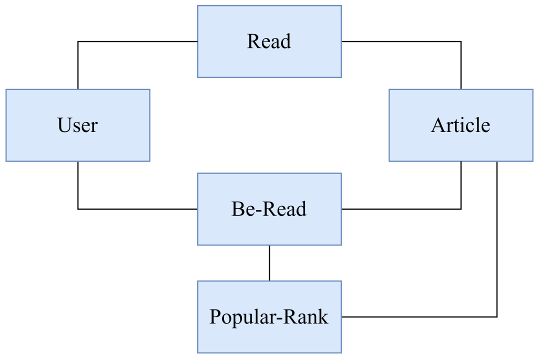


Figure 3: Structured data tables inter-relations.

User information is stored in the User table and article information is stored in the Article table, while the Read, Be-Read and Popular-Rank tables should regard the interactions between users and articles. Each table may be subject to fragmentation depending on a specific attribute from the table itself (horizontal fragmentation) or from a related table (derived horizontal fragmentation). The latter corresponds to the Read and Be-Read tables, since they depend on the fragmentation schemas of the Article and User tables, respectively. Furthermore, the partitioned data should be either distributed or replicated between two Database Management Systems (DBMSs). On a similar note, the unstructured data should also be stored in a distributed manner, accessible for queries regarding a given article.
In summary, one should implement the data center in a distributed context, with the following functionalities:

- Bulk data loading with data partitioning and replica consideration.
- Population of the empty Be-Read table, by inserting newly computed records.
- Query the top-5 daily/weekly/monthly popular articles details from the Popular-Rank table (text, image and video) (involving the join of Be-Read table and Article table, and the retrieval of the article contents from the distributed file system).
- Efficient execution of data insert, update and queries.
- Monitoring the running status of the DBMS servers, including its managed data (amount and location), workload, etc..

4 Design

We designed our solution taking into account the requirements of this DDBS project. Our final architecture consists of the following components:

- MongoDB Sharded Cluster
- Hadoop Distributed File System
- Redis Cache
- Tkinter App

The following figure describes the interaction of components within the architecture:

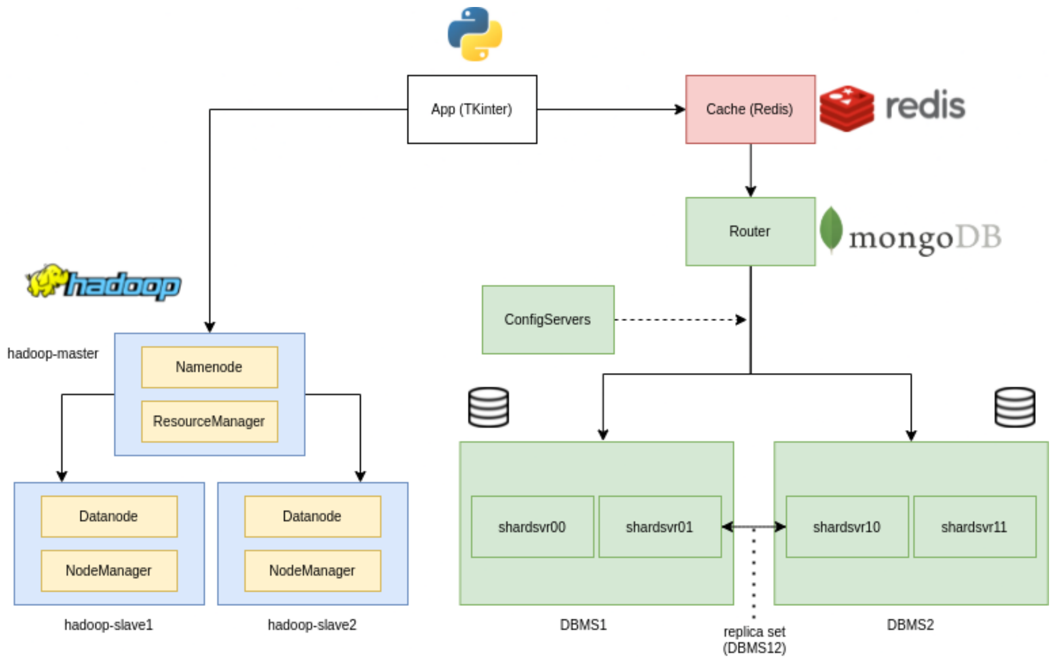


Figure 4: Solution architecture.

5 Implementation Details

5.1 MongoDB

In order to store 10.000 users, 10.000 articles and 1.000.000 reads we chose MongoDB as our Database Management System. We use MongoDB instead of MySQL, because although it might not be a relational database, it manages high amounts of data with high availability, with JSON oriented documents. With MongoDB Sharding, we were able to create two clusters of MongoDB servers, DBMS1 and DBMS2. Both shard clusters are displayed and can be interpreted as in Figure 5:

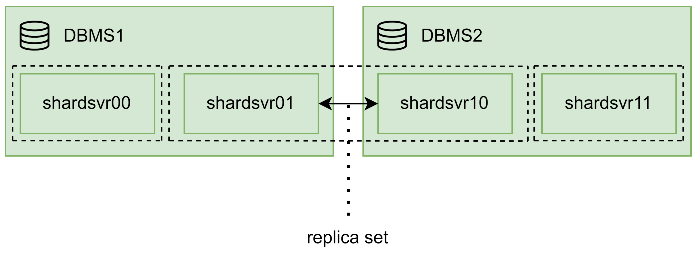


Figure 5: Shard arrangement.

We decided to have shards in the same replica set for each DBMS, so we can share particular data in both DBMS according the DDBS requirements, without having any distribution concerns. Moreover, we added another shard to each DBMS, because specific data may be store only in one of the DBMS. Also, we instantiate a replica set of two configuration servers, which will store the metadata of the sharded collection, knowing where the data is allocated, simplifying the flow of any query. In order to complete our sharding setting, the query router (mongos) will be the gateway for every connection to the sharded cluster. Operations will occur through the router, until the request is satisfied.
Since each of the DBMS could be geographically distant from each order, we could simulate this by using docker containers for each shard in the architecture. In this project we wanted to fragment our data, for instance based on the region of the user, in a real context it would be geographically convenient to have DBMS1 in Beijing and DBMS2 in Hong Kong. In order to fragment sharded

collections we tagged each DBMS and we created shard tags and specific ranges [6] for which the system could allocate the fragments, according the given requirements for all five collections.
Hereupon, we had to initialize all five collections of our sharded cluster. User, Article, and Read collections were simply populated with the given data generation files. On the other hand, Be-Read and Popular-Rank collections needed to be initialized based on the other three collections. To populate the Be-Read collection, we simply counted all the types of interactions between users and articles, adding the final calculated document to the sharded collection (one document per article). Furthermore, to initialize the Popular-Rank collection, we sort all articles by its number of reads for a specific day/week/month (timestamp key will identify each time span), after counting the number of reads for a specific period of time.
In order to simplify each insert/update/query to the database we created the class *QueryManager* with python. So, we defined the required and necessary queries to support the entire architecture, avoiding direct accesses to the router which could be unpleasant.

5.2 Redis

After the implementation of the MongoDB database, we also implemented another data storage layer with Redis. With this new cache layer we could store a subset of data, so that similar requests can be easily and quickly satisfied, that were previously retrieved or computed. Redis can cache key-value entries, so we decided to store documents with keys in the following format: <Collection Name> + <ID>. In order to optimize the cache usage, we decided to set LRU (Least Recently Used) [7] as the replacement policy when the cache is overwhelmed. As we set 100 megabytes as the cap for our cache, the LRU policy will force the removal of the least recently used element in the cache, every time this limit is reached. In addition, once a document is either deleted or updated on the MongoDB database, the cache is automatically notified with the most recent changes, so the cache coherency is guaranteed. Finally, we updated the python class *QueryManager*, which was created alongside with the MongoDB cluster, so that queries to MongoDB database will only take place when the cache is not storing some requested value.

5.3 Hadoop

Each article was generated with 1 text file and 1-3 images. Additionally, some of the articles also had 1 video in their content. In order to store the various unstructured data types, for a total of 10.000 articles, we decided to use Apache Hadoop. Apache Hadoop is an open source framework that, as previously mentioned, is used to efficiently store and process large datasets, namely composed by unstructured data. Instead of using one large computer to store and process the data, Hadoop allows clustering multiple computers to analyze the dataset in parallel and more quickly. Regarding its master/slave architecture, we determined there should be more than one slave node, with the aim of thoroughly experimenting the replication schema in the DataNodes, offered by HDFS. The article contents are then stored in the form of blocks, distributed and replicated through both of the slave clusters, assuring the authenticity of the data. Moreover, Hadoop YARN was also configurated, which provides us some very useful management and monitoring tools for the cluster nodes and resource usage. Accordingly, our final Hadoop architecture for dealing with the unstructured data is represented in Figure 6.

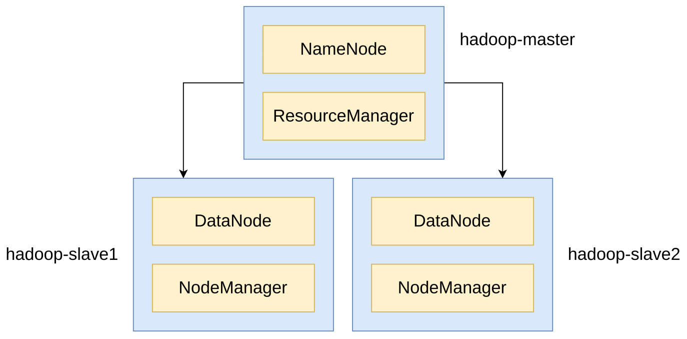


Figure 6: Hadoop arrangement.

Since we intended to fully simulate a distributed environment, we decided to run the master and each of the slaves on different Docker containers. Therefore, we relied on the WebHDFS REST API for the communication with the nodes. Finally, in favor of facilitating the integration of HDFS with the rest of our system, we developed a *HadoopManager* entity in python, which handles all the necessary interactions with WebHDFS (reading, uploading, downloading and deleting articles). This entity is then instanciated in the app with the purpose of dealing all the user requests, which are related to the unstructured data.

5.4 App (TKinter)

For integrating the developed data management components with each other, we decided to build a python application using TKinter. In the App, we implemented several menus which facilitate the interaction of the user with the diferent types of information stored in the data-center (users, articles and user-article interactions). For example, someone using the app is able to read, insert, edit and delete users at will, triggering the respective methods from the *QueryManager* entity. Moreover, similar functionalities are also possible for the rest of the structured tables. Finally, one may also access the contents of a given article directly from the application interface, since it will trigger the *HadoopManager* entity to retrieve the corresponding files from WebHDFS. Some of the features may be observed in Figures 7, 8, 9 and 10.

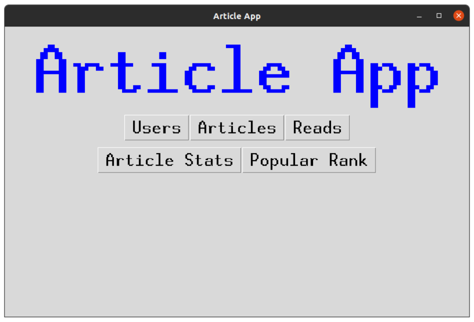
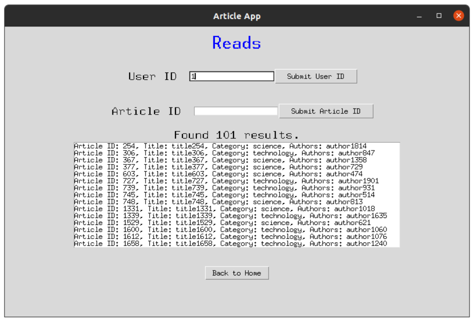
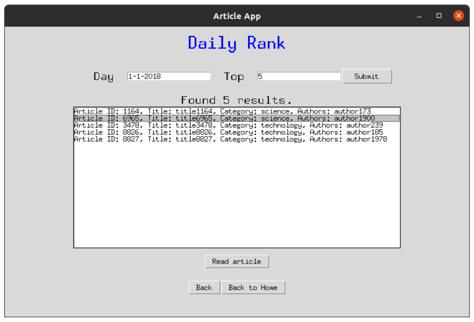


Figure 7: Application main menu.



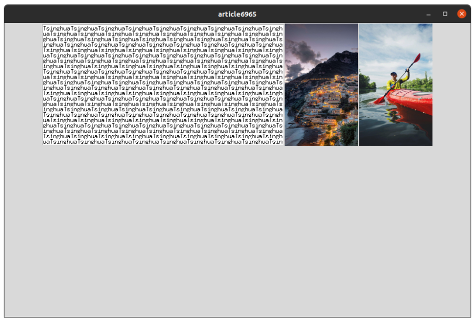
```
Found 101 results.
Article ID: 254, Title: title254, Category: science, Authors: author1814
Article ID: 306, Title: title306, Category: technology, Authors: author847
Article ID: 367, Title: title367, Category: science, Authors: author1358
Article ID: 377, Title: title377, Category: science, Authors: author729
Article ID: 603, Title: title603, Category: science, Authors: author474
Article ID: 727, Title: title727, Category: technology, Authors: author1901
Article ID: 739, Title: title739, Category: technology, Authors: author931
Article ID: 745, Title: title745, Category: technology, Authors: author514
Article ID: 748, Title: title748, Category: science, Authors: author813
Article ID: 1331, Title: title1331, Category: science, Authors: author1018
Article ID: 1339, Title: title1339, Category: technology, Authors: author1635
Article ID: 1529, Title: title1529, Category: science, Authors: author621
Article ID: 1600, Title: title1600, Category: technology, Authors: author1060
Article ID: 1612, Title: title1612, Category: technology, Authors: author1076
Article ID: 1658, Title: title1658, Category: technology, Authors: author1240
```

Figure 8: Articles read by user 1.



```
Found 5 results.
Article ID: 1164, Title: title1164, Category: science, Authors: author173
Article ID: 6965, Title: title6965, Category: science, Authors: author1900
Article ID: 3478, Title: title3478, Category: technology, Authors: author239
Article ID: 8826, Title: title8826, Category: technology, Authors: author185
Article ID: 8827, Title: title8827, Category: technology, Authors: author1978
```

Figure 9: Top-5 articles in a day (01/01/2018)



```
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
ghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTs
inghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghua
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
ghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTs
inghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghua
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
ghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTs
inghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghua
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
ghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTs
inghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghua
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
ghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTs
inghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghua
TsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsingh
uaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsinghuaTsin
```

Figure 10: Contents of article 6965.

6 Performance

In an attempt to optimize the performance of our designed DDBS, we used bulk write MongoDB functions so that data flow can be increased and processed faster. The initialization time decreases significantly while we use these MongoDB methods, which therefore will support bulk load operations. Also, introducing the Redis cache to our architecture really optimized queries that addressed big chunks of data, being remarkable the improvement with this extra data storage layer.
Additionally, one might consider that using the WebHDFS REST API has a slight negative effect on

the data transferring time, however, this decision was made based on our intention to fully represent a distributed context. Therefore, our system is completely prepared to run on separate and distant machines, which are not on the same network.

7 Conclusion

We were able to build a distributed database system which manages both structured and unstructured data, therefore, gaining a deep understanding of the advanced big data management techniques in a distributed environment (with the help of Docker containers). Moreover, we were also able to obtain some practical experience regarding the very latest big data technologies, such as MongoDB and Hadoop, by applying them to solve the proposed challenge.
However, additional improvements may also be integrated with our solution, for example, being able to add or drop DBMS servers at will, implementing data migration from one data center to others, and developing hot/cold standby DBMSs for fault tolerance. Furthermore, regarding the implemented solution structure, there are some changes which could enhance our performance, such as taking more advantage of the distributed context, by collecting statistics from the DBMS servers and optimizing the respective query and fragmentation schemes.
In summary, this project provided us a remarkable learning experience and we are excited to further improve our knowledge in the field, by developing some of the mentioned features on top of our current solution.

8 Work Contributions

David started the project by implementing the necessary tools to manage and process the structured data (MongoDB Sharding architecture design and python integration), while Armando was responsible for integrating and developing the unstructured data management tools (Hadoop DFS and YARN architectures design and python integration). Additionally, David also implemented a cache for the structured data queries (Redis). Both the elements worked together and cooperatively for the python application (TKinter), which interacts with all the previously mentioned components. Finally, the demo, report and manual resulted as well from a combined effort by Armando and David.

References

[1] AWS. *What is Big Data?* URL: `https://aws.amazon.com/big-data/what-is-big-data/`.

[2] SAS. *Big Data - What it is and why it matters*. URL: `https://www.sas.com/en_us/insights/big-data/what-is-big-data.html`.

[3] MongoDB. *Sharding*. URL: `https://docs.mongodb.com/manual/sharding/`.

[4] MySQL. *MySQL Cluster CGE*. URL: `https://www.mysql.com/products/cluster/`.

[5] Hadoop. *HDFS Architecture Guide*. URL: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[6] MongoDB. *Segmenting Data by Location*. URL: `https://docs.mongodb.com/manual/tutorial/sharding-segmenting-data-by-location/`.

[7] Redis. *Using Redis as an LRU cache*. URL: `https://redis.io/topics/lru-cache`.