CAP-Theorem in distributed databases
A comparison between Cassandra and MongoDB

Felix Myhsok
Department of Computer Science
University of Tromsø
fmy005@uit.no

Håkon Atlason
Department of Computer Science
University of Tromsø
hat000@uit.no

***Abstract*—Distributed databases like Cassandra and MongoDB are an essential part of today's internet infrastructure. They enable clients to access any data at any time and from any location. However, providing consistency, availability while maintaining partition tolerance goes against Brewer's CAP-Theorem. In this paper, we compare the two mentioned NoSQL storage systems based on their system architecture with a special focus on how they deal with the CAP-Theorem. We found that both systems have fundamental differences in their architecture which also impacts their way of dealing with the CAP-Theorem. Cassandra is, as a peer-to-peer system, designed to provide full availability while being partition tolerant but only enables eventual consistency. MongoDB on the other hand is a single master architecture developed for high consistency but lacks availability when performing a failover for the master. Regardless of their underlying design, both systems have the possibility to adjust the consistency level, gaining higher consistency or respectively better availability.**

***Index Terms*—CAP-Theorem, Distributed Database, Comparison, Cassandra, MongoDB**

I. Introduction

In NoSQL databases, in contrast to relational database management systems, data do not necessarily have to adhere to schemas, but are rather stored as values to corresponding keys, much like a hashmap. As these databases may be deployed in roles in which throughput or latency requirements keep rising, the need to expand the capabilities arises. The ability to increase the resource of the computers on which these databases run — scaling vertically — is, however, limited to the hardware available on the market. In this paper we aim to compare the two distributed NoSQL databases Cassandra and MongoDB, both of which support horizontal scaling, ie. increasing resources by connecting multiple computers together. In particular, we wish to compare them in light of the CAP-Theorem, which states that a partition-tolerant distributed system cannot guarantee both availability and consistency.

In Section II we introduce the CAP-theorem. We then give an introduction to Cassandra and MongoDB. In Section III we compare the cluster architecture of the two databases, how they approach the CAP-theorem, and compare their performance. Finally in Section V we present some related work before we conclude in Section VI.

II. Background

In this section we will introduce relevant background information about the CAP-Theorem and the two compared databases.

A. CAP-Theorem

The CAP-Theorem was first mentioned by Eric Brewer during a talk at PODC [5]. He put forward the thesis that a distributed system with shared data can never achieve **consistency**, **availability**, and **partition-tolerance** at the same time. However, all three properties are desirable and expected from a real-world application. Two years later, this theory was formally proven in the asynchronous network model by S. Gilbert and N. Lynch [7] and it was shown that only two out of the three properties can be accomplished simultaneously.

In the following, we will give a short overview of the three properties and the CAP-Theorem and define them before we show how they interact and depend on each other in distributed systems.

a) Consistency: Consistency means that there is no conflicting data in the system and all queries, regardless of their responsible node or point in time result in the same data. This usually requires a total order of operations and is equivalent to a system that runs on a centralized single instance.

b) Availability: Under availability is generally understood that for every request there is a response from the system. This includes that every request terminates even in case of network partitioning, however, an exact time is not specified. Only if the requesting node fails, a response might not be delivered to the client.

c) Partition Tolerance: If a system is partition tolerant, it can handle any network partitioning and thus the loss of an arbitrary number of messages without compromising the functionality or integrity of the system. Except in the case of a total network failure, any number of arising or disappearing partitions should not cause incorrectness of the system, especially with regards to the formerly mentioned properties consistency and availability. Network partitions occur if any two nodes cannot communicate anymore for any network-related reason. In today's internet topology, network partitioning is an often witnessed phenomenon and unavoidable in large-scale systems.

B. Cassandra Database

Cassandra is a column-based, distributed database for key-value pairs and was originally developed by Facebook for their message inbox search. We base our work on their original publication about their general architecture and design published in 2010 [12]. Today, Cassandra is part of the open-source Apache Foundation [6] and has undergone many revisions as well as extended by multiple features which we consider out of scope for this comparison.

The main motivation behind the original development of Cassandra was to create a system that can store large amounts of unstructured data on commodity servers while allowing continuous horizontal scalability[1]. The system is based on a peer-to-peer architecture where all the nodes have equal rights and abilities. The nodes are structured in a fixed circular space (”ring”) which is based on the output range of a consistent hashing function. The general architecture of Cassandra is visualized in Figure 1a. Each node is assigned a value within this space which represents its position and determines the keys that are stored on that node. To ensure that data is distributed evenly across the nodes, Cassandra allows load balancing by moving the position of the nodes on the ring. Within this peer-to-peer system, a node is aware of all the other nodes and knows which key ranges they maintain. This information is shared with the gossip protocol which is also utilized to identify offline nodes.

Depending on the individual configuration, the key-value pairs are stored on multiple nodes. The selection of nodes is based on the chosen replication strategy, ranging from Rack-unaware to Datacenter-aware. Besides the replication strategy, the system is configurable either for synchronous or asynchronous replication.

On a single node, the data is written to a commit-log for maximal throughput and recoverability and an in-memory data structure. The in-memory data structure is eventually written to an immutable file and maintained by a background process.

C. MongoDB

MongoDB was first released in 2009, and is an open-source distributed document-oriented NoSQL database (sometimes called a document store) [13]. It stores documents in the JSON-like format BSON, with the documents being organized into collections [14]. Schemas are not required, but can be enforced. However it is not a relational database — it is designed to contain denormalized data [8].

MongoDB uses a master/slave architecture[2], and can be deployed in a configuration with one master, called the primary, and multiple slaves, called secondaries. A primary and its secondaries are referred to collectively as a replica set. The replica set serves to store one set of data, which is dictated by the primary and replicated on the secondaries. In this configuration, a client can connect directly to the primary and perform operations. In the event that the primary goes down, a new primary will be elected, thus achieving fault tolerance.

Alternatively, in order to scale horizontally[1], multiple replica sets can be deployed together, with data partitioned across them; this is called a sharded cluster. In a sharded cluster, the replica sets are referred to as shards, and data are partitioned across these. Each shard has a primary and its own secondaries. Added to the cluster as well are other components to handle the work needed to maintain the shards.

[1]Adding more machines to the cluster

[2]MongoDB can also be deployed in a standalone configuration, but this is mainly for development or experimental purposes[13]

III. Comparison

In this section we compare the key aspects of the two databases.

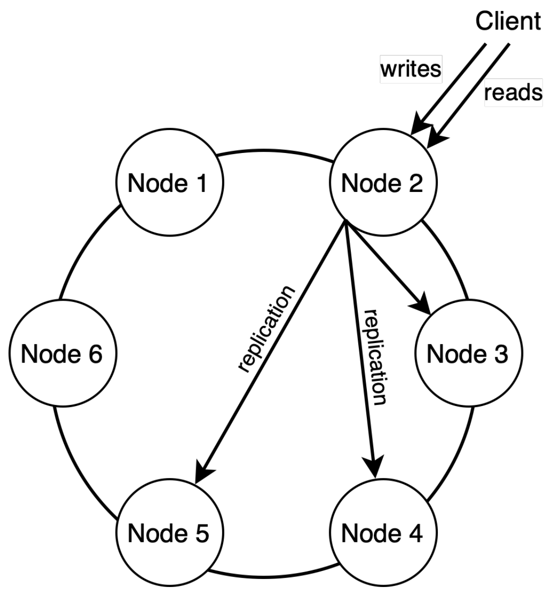
A. Architectures

Cassandra and MongoDB have fundamentally differing cluster architectures; at the core, they feature decentralized and centralized architectures, respectively. In this subsection, the architectures of non-sharded MongoDB, sharded MongoDB, and Cassandra databases will be compared.

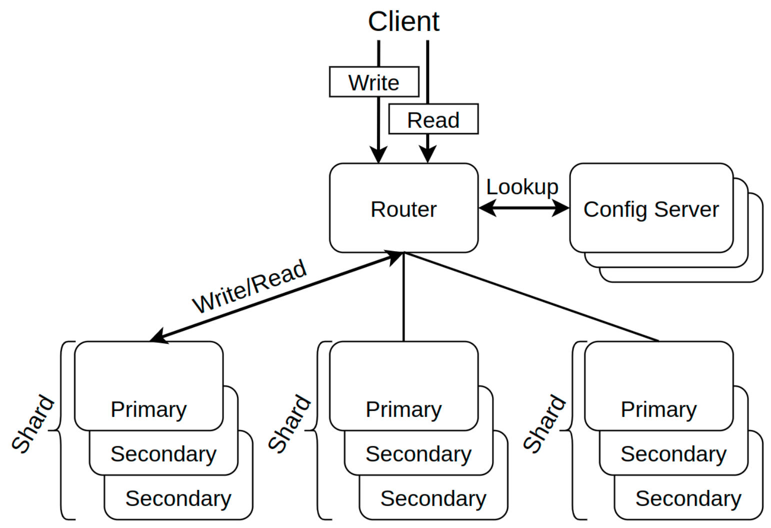
Within the context of MongoDB, every server within a replica set, ie. a group of servers that maintain the same data, will be running an instance of MongoD (Mongo Daemon). The primary maintains an operation log (oplog), to which it writes all transactions. The rest of the nodes, ie. the secondaries, which serve to replicate the primary’s data, execute the operations from the oplog to replicate the changes. Members of a replica set also exchange ”heartbeat” messages, to maintain updated information about the operativity of each other. In the event that the primary loses connection to more than half of its secondaries or becomes wholly unavailable, a primary election takes place via a RAFT protocol [8]. One arbiter node may be added to the cluster to this end; it takes part in leader elections, but does not store any data. The number of secondaries is dictated by the replication factor. A higher replication factor results in higher fault tolerance.

To scale MongoDB horizontally, a sharded cluster can be employed (see figure 1b). In a sharded cluster one or more collections are distributed across two or more replica sets [2], called shards, such that each shard maintains a subset of keys. While this can increase the time to execute any one operation, it can increase the over-all throughput [8], which is the goal. How the data is distributed across the cluster is dictated by the shard key, which consists of one or more fields. In this configuration, the client sends requests to a request router, which is a MongoS instance. Each client may maintain its own request router [11]. The cluster will also contain a configuration server, which has replicas of its own. The configuration server contains information about how data is distributed across shards [8], and communicates this to the router.

In both a non-sharded and a sharded configuration, consistency can be sacrificed for higher read throughput. This is achieved by adjusting read preference [8], which is decided by the client. The client can decide to allow reads from secondaries [9], which results in eventually consistent behavior,



(a) Architecture of a Cassandra cluster



(b) Architecture of a MongoDB sharded cluster

Fig. 1: System Designs

rather than strongly consistent. Relatedly, write concern, also dictated by the client, dictates how many secondaries need to replicate a write operation in the primary before the write is considered complete. This means that writes are less likely to fail after having been issued.

A Cassandra cluster is a peer network, which means that the failure of any one node will not prevent the system from responding, given that the client can reach another node in the cluster. Cassandra employs a ring architecture, and uses a consistent hashing function to split the responsibility for the keys around the nodes; this node is called the coordinator for the key. Data is also replicated on other nodes, as dictated by the replication factor, and all nodes keep a cache of the ranges of other nodes. With the simplest replication policy, ”rack unaware”, the replicas are the successor nodes of the coordinator node, while the ”rack aware” and ”datacenter aware” policies involve more involved algorithms [12]. However, it should be noted, that a node acts only as a coordinator when serving a request, and it does not matter which node serves the request, as long as the node contains the key queried. Upon joining a cluster, what data should is replicated on which nodes may change; in this case, the split chosen will be biased towards splitting the keys such that nodes already containing a range keep that range, in order to minimize data transfers.

The fundamental architectural difference then, is that MongoDB features a strong hierarchy, in both sharded and non-sharded configurations, while Cassandra employs a peer network. By default, any Cassandra node can service any request (rerouting it as needed), while in a MongoDB the client may only direct its requests to the MongoS router, which will then use the configuration server to lookup what shard(s) to forward the request to. Furthermore, in a MongoDB cluster, each component has a specific role and run specific program instances, in contrast to Cassandra, where every node runs the same program. It stands to reason that a peer-to-peer network lends itself well to an available system, as while a decentralized structure may make consistency more demanding to implement, it means the client may route its request to any node, and get a response. On the other hand, a centralized architecture seems to suit a consistent system, as it is always clear what node can be expected to contain consistent data. Common for the two databases is that they do not — or can be configured such that they do not — contain any one single point of failure (barring client failure).

B. *CAP-Theorem*

As introduced in Section II-A, according to CAP-Theorem a distributed system can’t achieve consistency, availability, and partition tolerance at the same time. However, all 3 properties are desirable in a distributed database. In this section, we will compare how the two databases approach this challenge and where they set their focus.

a) Partition Tolerance: Cassandra and MongoDB are both distributed databases that run potentially on an arbitrary number of nodes spread across any network. Since network partitioning is likely to occur at any time both systems must be partition tolerant. To keep responding even when nodes fail or messages are dropped or delayed is also a core requirement for distributed systems in general and thus build into the foundation of Cassandra and MongoDB.

b) Consistency: When comparing the databases for general consistency, they show fundamental differences.

Based on its default architecture MongoDB is strongly consistent. Once a write is successfully acknowledged by a primary node, all subsequent reads from the primary will return this value since the primary node is in this case a single point of truth. However, it is also possible for the client to allow executing reads from secondaries. While this may come with an improvement in latency, as it alleviates load from the primary [8], this weakens the consistency property since the secondaries might contain stale data for any reasons. Since it is unclear at which point in time the secondaries will reflect the same data as the primary, this consistency level is called eventually consistent.

Cassandra is from the ground up an eventually consistent system. The node responsible for the key returns success once it has written the data to disk. The other replicas will eventually get the data, leading to potentially inconsistent reads from these replicas.

Besides their default design, Cassandra and MongoDB offer additional configuration levels for consistency. Both databases use therefore a similar approach: Depending on the chosen consistency level, a certain amount of secondaries/replicas needs to acknowledge the data back to the primary/responsible node before the data is acknowledged to the client. The chosen amount of replicas can range from 0 to all possible replicas. This ensures that the data is written to enough replicas and thus increases the consistency but comes at the cost of weakened availability in case of node failures and increased latency. With this approach, Cassandra can provide strong consistency if the sum of nodes where the data is written to and read from is greater than the replication factor.

While MongoDB provides no option to update secondaries that are lagging except to wait for them to catch up, Cassandra has two features to update inconsistent data: Read repair and hinted handoff. When a node receives inconsistent data from multiple replicas during a read operation, it always chooses the data with the greatest timestamp to ensure its the most recent value (read repair). Hinted handoff helps to limit the impact of temporary failures. If a replica is not reachable, the node that coordinates the data replication stores the failed writes temporarily as hints and re-sends them once the node is back online.

c) Availability: Both NoSQL storage solutions also show significantly different approaches to providing availability. Generally, both solutions increase their availability with additional replicas.

In a Cassandra cluster, all the nodes are equal peers and the data is replicated on multiple nodes. This allows the system to simply switch over to another node in case a node fails without any delay or downtime. As long as the number of failures does not surpass the specified consistency level and replication factor, the system can respond to all requests and is thus fully available. Further, since a client can send requests to any node in the cluster, also a network partitioning where a client can't communicate to one specific node, doesn't compromise the availability as long as the nodes can communicate with each other.

Since MongoDB is built as a single master architecture, the availability of the system strongly depends on this node. If the primary node becomes unavailable, the other secondaries will elect a node as a new leader. This failover takes about 12 seconds with the default settings [10]. During this time MongoDB suspends all writes and is thus not available. Reads from secondary nodes, if permitted by the client, are possible as long as the majority of secondaries are reachable and the configuration permits it. Only after the election of a new primary, the system gains full availability again. In contrast to Cassandra, this failover is necessary as soon as the client cannot reach the primary node which can also be caused by a network partitioning.

Cassandra and MongoDB have both the ability to further decrease the likelihood of unavailability by replicating into multiple geographically separated data centers.

C. Performance

In 2013 Ambramova & Bernardino [1] conducted a set of six experiments comparing MongoDB and Cassandra running on one node. The experiments generally showed that in read-heavy workloads, MongoDB performed better for a small database (100MB), while Cassandra outperformed MongoDB as the size of the database grew (280MB and 700MB). In a 50/50 read/write workload, Cassandra achieved significantly higher (53-61%) throughput at all sizes. In two update-heavy experiments, Cassandra also achieved significantly higher (93% and 97%) throughput averaged across all sizes.

In real-world deployments, however, data is likely to be replicated, to aid in fault tolerance and availability. In a 95/5 read/write experiment (with uniformly randomly selected records) Haughian et al. [9], compared the overhead of setting replication factor to 2 (ie. one replica), compared to a replication factor of 1. The overhead for a 3-node Cassandra cluster was measured to 1.5% . For a 3-node MongoDB non-sharded MongoDB cluster (with reads allowed from secondaries) the overhead was 94.1%. For both clusters consistency level was set to have only one node needing to reply to a request. When setting write concern to requiring a quorum (a majority), but otherwise same configuration, the numbers Cassandra measured an 8.6% overhead, while MongoDB measured a 95.4% overhead. For the 5/95 read/write experiment, the corresponding figures were Cassandra 61.3% versus MongoDB 120.4% for write concern set to 1, while for a quorum required the numbers show Cassandra 70.5% versus MongoDB 99%. These findings suggest that the relative overhead brought by higher replication factors is significantly lower for Cassandra than for MongoDB. However, one issue is that since writes in MongoDB were allowed from secondaries, the system no longer guarantees strong consistency. The paper also presents numbers for larger cluster sizes (up to 12 nodes), but as the replication factor differs at cluster sizes above 3, these results will not be presented here in detail, though in general they seem to favor MongoDB. This is possibly due to the aforementioned discrepancy in replication factor.

On average across all experiments (including the experiments with unequal replication factor) it was shown [9] that MongoDB achieved on average 5.1% higher throughput than Cassandra in read-heavy workload (95/5 read/write). While

Cassandra achieved 72.5% higher throughput than MongoDB in write-heavy workload (5/95 read/write).

These studies have short-comings, but if we accept these, the take-away is that Cassandra generally manages moderately to significantly higher throughput for write-heavy workloads, while also managing to stay competitive in read-heavy workloads.

IV. Discussion

In most experiments studied during the writing of this paper [1, 9, 4, 3], Cassandra generally stays on par or outperforms MongoDB. Especially so in experiments featuring write-heavy workloads, where Cassandra most often achieves higher throughput. The experiments also suggest that Cassandra might scale better horizontally than MongoDB.

However, it should be kept in mind that, as mentioned, Cassandra is a column-based store while MongoDB is a document store. One consequence of this is demonstrated by Baruffa et al. [3], who compared processing time for a radio spectrum surveillance service when using Cassandra and MongoDB as a database; the queries written for each database are very different and, as is often the case, there likely would have been multiple valid ways of writing queries for both databases. This raises the question of how one, in order to fairly compare performance of two different NoSQL databases, would write a benchmark that does not favor one database, when simulating real world workloads, and if it should even be attempted.

Certainly, a database should not chosen based on performance alone, as there are many factors that affect the viability of deploying a given database in a given situation. Some examples are proficiency with the system, available hardware resources, network latency, expected workload, and last but not least, its relation to the CAP theorem. The value of strong consistency or availability is not inherently measurable.

All that to say databases have different use cases, and comparing databases that have different approaches to the CAP-Theorem necessarily will be an apples-to-oranges comparison. A social networking site, for instance, when faced with the choice between having eventually consistent data, or having unavailable data, the former choice may be well justified, as a drop in users may cause a drop in advertisement revenue. Another firm, for instance a financial institution, might want strong consistency when handling data detailing financial transactions. The risk of having services become unavailable might be preferable to the risk of having inconsistent data written to its servers.

V. Related Work

Since the initial releases of both databases (Cassandra in 2008 and MongoDB in 2009), a variety of studies utilized and compared them in different applications, such as G. Baruffa et al. [3] for monitoring the radio spectrum.

Since Cassandra and MongoDB are among the most popular NoSQL databases, there also have been multiple direct comparisons to highlight similarities and differences.

One example of such a comparison is the paper published by Veronika Abramova and Jorge Bernardino [1]. After a very comprehensible introduction to NoSQL databases and their properties in general, they compared the features of both architectures from a high-level perspective. Furthermore, they executed six different workloads on each database and analyzed their execution times. They concluded that Cassandra provided better (faster) results for almost all scenarios. Especially with increasing data size, MongoDB was losing performance.

A similar approach is described by C. Băzăr et al. [4]. They compared the tree distributed databases Cassandra, MongoDB, and Couchbase from a high-level perspective and put special emphasis on the key criteria of NoSQL storage solutions: Scalability, Performance, Availability, and Ease of development. Additionally, they surveyed their performance by simulating an interactive application and compared the latency for reads and writes introduced through increased throughput. While Cassandra and MongoDB showed very similar performance for reads, Cassandra was able to achieve a lower latency for writes. However, Couchbase outperformed both of them in read as well as in write latency independent of the throughput.

G. Haughian et al. [9] benchmarked Cassandra and MongoDB with eight different workload configurations and analyzed their throughput, the access distribution, and especially the impact of the replication. In accordance with [1, 4] they found that Cassandra performs well with write-heavy workloads but increasing replication has a significant impact on the performance. MongoDB on the other side outperforms with read-heavy workloads and the impact of additional replication is only marginally. They concluded that the single-master architecture of MongoDB provides significant benefits when it comes to replication in comparison with the multi-master architecture of Cassandra.

Even though there already have been multiple comparisons between Cassandra and MongoDB, none of them focused on their fundamental architecture differences regarding the CAP-Theorem, which is subject to this study.

VI. Conclusion

Distributed databases are an essential part of today's internet infrastructure. From the client's perspective, these services need to provide the correct data and be reachable at all times from any location around the world. However, Brewer's CAP-Theorem states achieving these three properties (consistency, availability, and partition tolerance) at the same time is not possible in a distributed system.

We compared two of the most popular open-source, NoSQL storage solutions, Cassandra and MongoDB. Besides analyzing the underlying architecture, we focused on how these distributed databases encounter the CAP-Theorem. Both systems are based on fundamentally different architectures.

While Cassandra deploys a fully-fledged peer-to-peer network based on consistent hashing, MongoDB's design follows the single master principle where additional replicas are stored on slaves. These architectural differences also lead to different approaches regarding the CAP-Theorem. Cassandra can provide availability while being partition tolerant but therefore can only offer eventual consistency. Indifference, MongoDB is highly consistent but temporarily loses its availability when the client cannot reach the primary until a new master node is elected. Besides these different approaches, both offer the availability to tune their consistency level for read and write operations, trading availability for consistency, or the other way around. This allows configuring the systems in a way that they have highly similar properties in regards to the CAP-Theorem.

In summary, are both highly functional storage solutions and have a significant overlap in their capabilities. However, in certain scenarios and edge-cases the systems guarantee different properties resulting from their underlying architecture.

Since both systems are continuously developed, future work should focus on the comparison of new features. Besides Cassandra and MongoDB, there exists a large variety of other distributed databases which also could be analyzed to further understand how to efficiently work with the CAP-Theorem.

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*. 2013, pp. 14–22.

[2] Jaumin Ajdari and Brilant Kasami. "MapReduce Performance in MongoDB Sharded Collections". In: *International Journal Of Advanced Computer Science And Applications* 9.6 (2018), pp. 115–120.

[3] Giuseppe Baruffa et al. "Comparison of MongoDB and Cassandra databases for spectrum monitoring As-a-Service". In: *IEEE Transactions on Network and Service Management* 17.1 (2019), pp. 346–360.

[4] Cristina Băzăr, Cosmin Sebastian Iosif, et al. "The transition from rdbms to nosql. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase". In: *Database Systems Journal* 5.2 (2014), pp. 49–59.

[5] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. 10.1145. Portland, OR. 2000, pp. 343477–343502.

[6] The Apache Software Foundation. *Open Source NoSQL Database*. Last accessed 30 March 2022. 2022. URL: https://cassandra.apache.org.

[7] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[8] Guy Harrison and Michael Harrison. *MongoDB performance tuning: Optimizing MongoDB databases and their applications*. Apress, 2021.

[9] Gerard Haughian, Rasha Osman, and William J Knottenbelt. "Benchmarking replication in cassandra and mongodb nosql datastores". In: *International Conference on Database and Expert Systems Applications*. Springer. 2016, pp. 152–166.

[10] MongoDB Inc. *Replica Set Elections*. Last accessed 30 March 2022. 2021. URL: https://www.mongodb.com/docs/manual/core/replica-set-elections/.

[11] MongoDB Inc. *Sharded Cluster Components*. Last accessed 17 April 2022. 2021. URL: www.mongodb.com/docs/manual/core/sharded-cluster-components/#number-of-mongos-and-distribution.

[12] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[13] Amit Phaltankar, Michael Harrison Juned Ahsan, and Liviu Nedov. *MongoDB fundamentals: A hands-on guide to using mongodb and Atlas in the real world*. Packt Publishing, 2020.

[14] William Schultz, Tess Avitabile, and Alyson Cabral. "Tunable consistency in mongodb". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2071–2081.