San Jose State University

SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 5-8-2020

Developing a MongoDB Monitoring System using NoSQL Databases for Monitored Data Management

Anjitha Karattu Thodi
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Databases and Information Systems Commons

Recommended Citation

Thodi, Anjitha Karattu, "Developing a MongoDB Monitoring System using NoSQL Databases for Monitored Data Management" (2020). *Master's Projects*. 909.
DOI: https://doi.org/10.31979/etd.sh3j-uwtp
https://scholarworks.sjsu.edu/etd_projects/909

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Developing a MongoDB Monitoring System using NoSQL Databases for Monitored Data Management

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Anjitha Karattu Thodi

May 2020

© 2020

Anjitha Karattu Thodi

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Developing a MongoDB Monitoring System using NoSQL Databases for Monitored Data Management

by

Anjitha Karattu Thodi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2020

Dr. Suneuy Kim	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

Developing a MongoDB Monitoring System using NoSQL Databases for Monitored Data Management

by Anjitha Karattu Thodi

MongoDB is a NoSQL database, specifically used to efficiently store and access a large quantity of unstructured data over a distributed cluster of nodes. As the number of nodes in the cluster increases, it becomes difficult to manually monitor different components of the database. This poses an interesting problem of monitoring the MongoDB database to view the state of the system at any point. Although a few proprietary monitoring tools exist to monitor MongoDB clusters, they are not freely available for use in academia. Therefore, the focus of this project is to create a monitoring system that is completely built from open-source resources. To automatically monitor a MongoDB cluster, several components are to be built: monitoring agents that obtain this information from the nodes in the cluster, storage mechanisms to save this information for future use and write buffers to temporarily hold monitored records before they are written to storage. The monitoring agents have to be created to obtain only the information that a user of a monitoring system might find useful. Since monitored data is expected to be of high volume and velocity, NoSQL databases are ideal candidates for the storage component of the monitoring system. MongoDB, Cassandra, and OpenTSDB are identified as suitable candidates and their performances are compared with respect to several aspects such as read and write performance and storage requirements. In an attempt to improve the write performance of the system, the performance impact of adding a BigQueue as a write buffer to the storage is also studied.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. Suneuy Kim for her guidance and support throughout the duration of the project. Her advice and motivation have been pivotal in the completion of my work. I also thank my committee members Dr. Thomas Austin and Dr. Robert Chun for their input and feedback, which were extremely helpful in enhancing the project. Finally, I thank my family and friends for their never-ending support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Over the past two decades, the importance of data has risen dramatically to the point that it is now considered to be a valuable commodity. A tremendous number of fields and technologies rely on data to function effectively. In Artificial Intelligence, machine learning and deep learning models require large amounts of data for machines to learn to identify patterns and make predictions based on them. In advertising, data is extremely crucial in targeting advertisements based on user preferences. Data is also continuously generated in large volumes every day. IoT devices like smartwatches, smart TVs and home assistants are seen in most homes today and generate data of different formats. Similarly, different kinds of sensors continuously produce data every second or minute of every day. Therefore, it is necessary to be able to effectively store and retrieve such large and varied data, also known as “Big Data”.

While relational databases have traditionally been used for reliable persistence of structured data, they pose several disadvantages when it comes to dealing with big data. Since big data is commonly expected to be unstructured or loosely structured, the rigid schema requirements of relational databases make it difficult to store big data. Moreover, as the amount of data grows, relational databases are unable to scale accordingly. These shortcomings of relational databases concerning big data have led to the popularity of NoSQL databases [7]. NoSQL databases are highly flexible in the type of data that they can store, and are ideal for storing unstructured and semi-structured data in large quantities. NoSQL databases are designed to run on a cluster of nodes and support horizontal scalability by evenly distributing data across multiple nodes in the cluster. From the perspective of users, these distributed databases still behave as if the data is stored at a single location. This ensures the simplicity of basic operations such as reads and writes. Most NoSQL databases also ensure reliability by replicating data across different nodes. Some of the most popular NoSQL databases include MongoDB, Cassandra, HBase, Couchbase and Redis [8].

Distributed database systems are more difficult to maintain as compared to traditional relational databases running on a single server. The administrators of distributed databases have to simultaneously keep track of each node in the cluster. Almost all NoSQL databases provide several commands and tools that can be used to display different metrics from the database. However, it is tedious and often downright impossible to monitor all components of a large NoSQL database manually. Monitoring systems can be developed to automatically monitor predefined metrics from each node in the cluster and display these values to the users in real-time. At every instant, a monitoring system should be able to provide an accurate summary of the current state of the system. This eliminates the need for manually looking up metrics and statistics in each node using multiple commands and greatly simplifies database administration.

In this project, a system for monitoring MongoDB clusters has been created. Extensive research was performed to select each component of the monitoring system to ensure its performance and correctness. Due to the immense usefulness and convenience of monitoring systems, a lot of companies have built their versions of monitoring systems for NoSQL databases including MongoDB. However, these are not released for general use and are built according to the companies' specific use cases. The goal of this project has been to build a monitoring system using only open-source tools that can be accessed by individuals in academia and to capture all information that may be useful to a general user. The focus of the project is to ensure that the developed monitoring system is built with components that offer the best utility and performance to meet the goal of this project.

This project report is organized as follows: Chapter 2 provides an overview of the fundamental concepts related to this project. Chapter 3 surveys literature and techniques related to the project. Chapter 4 overviews the concept of using a BigQueue as a write buffer and possible implementations of BigQueue. The proposed design and implementation details of the monitoring system built in this project are explained in Chapter 5, along with the rationale behind the design decisions. Experiments are conducted to evaluate the best components and design elements to implement the monitoring system. Chapter 6 presents the experiment setup, experiment results, and analysis. Finally, Chapter 7 concludes the paper and briefly describes future work.

CHAPTER 2

Background

2.1 NoSQL Databases

NoSQL databases emerged as innovative data management solutions for big data. To understand why NoSQL databases are designed as they are, it is important to understand the characteristics of big data. As the name suggests, “big data” stands for data in large amounts. “Big Data” as a field of study deals with the storage, access, and analytics of big data. NoSQL databases are often discussed in the context of Big Data as it is the most common way to store, retrieve and manage big data. The three characteristics commonly used to describe big data, also known as the “3 Vs” of big data are [9]:

1. Volume: Largeness of the datasets. For example, a “big” dataset could be petabytes in size.
2. Velocity: The speed at which data is generated. For example, IoT devices continuously produce data at a high speed and are said to have a high velocity.
3. Variety: Variance in the type of data. For example, the data may be in the form of text, images, audio, etc.

Relational databases enforce all transactions to follow ACID properties - atomicity, consistency, isolation, and durability. While excellent for storing and querying structured data and for ensuring reliability, they are ill-equipped to handle big data with the above characteristics. As the “volume” of the dataset grows, a relational database can only attempt to grow vertically by adding more resources (e.g.: CPU, RAM) to the same machine. However, since such a database sits on a single machine, there is a limit to how much vertical scaling can be performed. If the data is generated at a high “velocity”, the single node hosting the relational database will be overloaded with writes, causing a bottleneck. Also, relational databases are characterized by a predefined and rigid schema, which cannot handle “variety” in data. Hence, relational or SQL databases are not a good fit to store big data.

NoSQL databases address the above issues by relaxing many of the constraints that were enforced by relational databases. The biggest difference is in how scaling is handled. NoSQL databases are designed to work on distributed clusters of multiple nodes rather than just a single node. This provides several advantages - the ability to store large datasets by splitting them across multiple nodes, load balancing by distributing the traffic across different nodes, etc. Sharding and replication are prominent data distribution techniques for NoSQL systems.

The core of horizontal scaling lies in splitting the dataset into pieces and distributing these pieces across different nodes in the cluster. This process is called sharding and the “pieces” are referred to as shards. Sharding contributes to scalability and improves write performance. After sharding a dataset, there is a choice to store each shard in just one node or to replicate the shard by storing multiple copies across different nodes. If the former is chosen and the shards are not replicated, a node failure could mean that access to the shards in that node is lost. However, if replication is chosen, each shard will be persisted in multiple nodes (depending on the replication factor). Replication is used to provide durability and availability, and improves read performance. In case a node containing a copy of the shard fails, the other nodes will still contain copies of the shard. There are two models of replications:

- Master-Slave replication: If a shard is replicated in x nodes, one node is the master or primary node for that shard and the remaining x-1 nodes are the slave nodes. All writes are directed to the master node, but reads can be performed on any node.
- Peer-to-Peer replication: All nodes containing the replica have the same role,

and there is no master. All nodes can serve reads and writes.

The powerful combination of replication and sharding ensures scalability and durability. However, it appears that issues with consistency could arise if all replicas do not always contain the most recent copy of the data. In most NoSQL databases, consistency is guaranteed as eventual consistency: eventually, all replicas will be in sync, but there may be windows of inconsistency in between.

Lastly, it is worth to note that there are four different types of NoSQL databases [10]:

1. Key-Value Store: Data is stored similar to a map or a key-value pair. Keys typically contain the name of the "fields" and "Values" contain the corresponding value of the field. e.g.: Redis, Riak

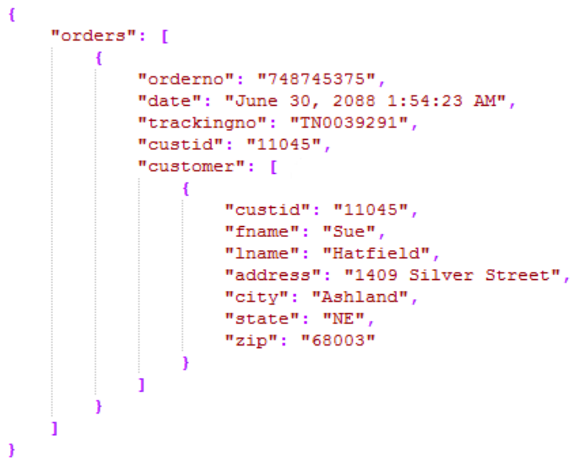
2. Document Database: In a document database, related information is stored in structures called documents. e.g.: MongoDB

3. Column and Column-family Oriented Database: Related information is stored in the same column or column family (a collection of columns). e.g.: HBase, Cassandra

4. Graph Database: Nodes and edges are used in graph databases to represent entities and their relationships. e.g.: Neo4j

2.2 MongoDB

MongoDB is a document-oriented NoSQL database. It has steadily been gaining popularity due to its great functionality and ease of use. It is currently ranked as the top choice for NoSQL databases according to most rankings including DB-Engines and KDNuggets. [1] describes a MongoDB document as an ordered collection of key-value pairs where the keys are strings and values can be objects like integers, strings or other documents. MongoDB represents its data as JSON (JavaScript Object Notation) documents. Figure 1 shows a simple example of a JSON document that stores order details. A group of related MongoDB documents is called a collection. For example, in a MongoDB collection named “Orders” that stores all orders in a shop, each document would contain details about a specific order. In relational terms, a collection is equivalent to a table and a document to a row within the table [1]. Collections, in turn, are grouped into databases.



```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

Figure 1: Example of a JSON document.

When a document is embedded or nested within another document, it is called an Embedded document [11]. It is particularly useful to denormalize data by storing multiple related documents in a single document as it improves query performance. Some other important terminologies associated with MongoDB are:

- mongod: mongod is the main process of MongoDB that runs in the background and performs all majors tasks including management and handling requests. A mongod runs on each node running MongoDB. [2]
- mongos: MongoDB is designed to run on a cluster of multiple nodes through the concept of sharding. Data is sharded and distributed to different nodes in

the cluster. mongos provides a layer of abstraction between the user and the sharded cluster by acting as an interface between them, as shown in Figure 2. When writes and reads are performed on data in a MongoDB cluster, mongos routes them based on the sharding information. In contrast, Mongodb that is not sharded is shown in Figure 3.

- config server: All metadata regarding the sharded cluster is stored in the config server. This includes information such as the distribution of data across shards.

2.2.1 Sharding in MongoDB

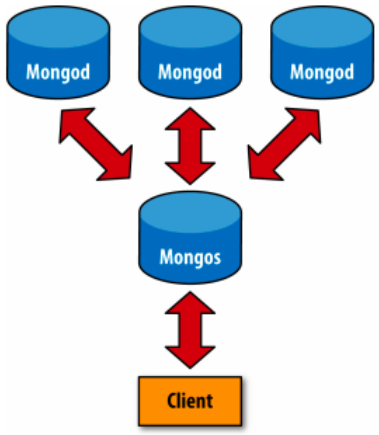


Figure 2: Sharded MongoDB Cluster [1].

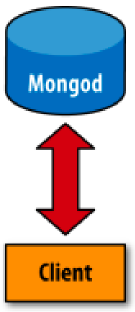


Figure 3: Non-sharded MongoDB [1].

MongoDB is designed to work on a multi-node cluster and can be configured for sharding and replication. MongoDB performs auto-sharding which means that once the user configures relevant settings, MongoDB takes care of the operations relating to sharding. This allows the user to use the database as if it were running on a single-node, simplifying the task of the user as compared to manual sharding. If the user chooses to shard a collection, they must specify the shard key, which is the field or combination of fields that the partitioning is based on. The documents with the same shard key value are grouped into one chunk. The selection of the shard key has a huge impact on the performance of the system and the distribution of data chunks across the nodes. For example, if a field with low cardinality such as gender is chosen as a shard key, we end up with a few large chunks which cannot be split further. To prevent this issue, a common way is to choose compound shard keys, which are shard keys consisting of more than one field. The user can also choose between two sharding strategies:

- Ranged sharding: In ranged sharding, data is divided into chunks based on the value of the shard key. For example, if "Age" is the shard key for a collection, documents with Age value from 1 to 30 may go to Shard A, 31 to 60 to Shard B and 61 to 90 to Shard B. Figure 4. shows range-based sharding. In ranged sharding, there is a risk of "hotspots" if the majority of the documents coming in fall within a particular shard range. For example, if most of the documents are for ages 31 to 60, Shard B runs a risk of hot-spotting.

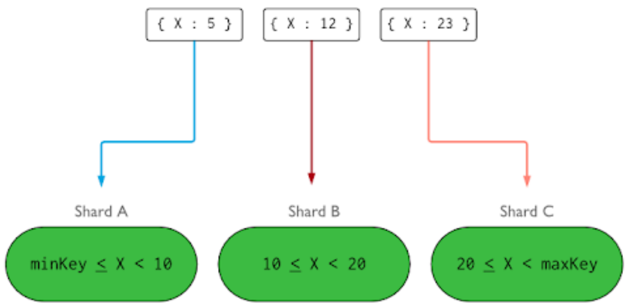


Figure 4: Range sharding [2].

- Hashed sharding: In hashed sharding, a hash of the shard key values is used to partition the data. This results in a more even distribution of data across shards and is a solution to the risk of hotspots. Figure 5. demonstrates hashed sharding.

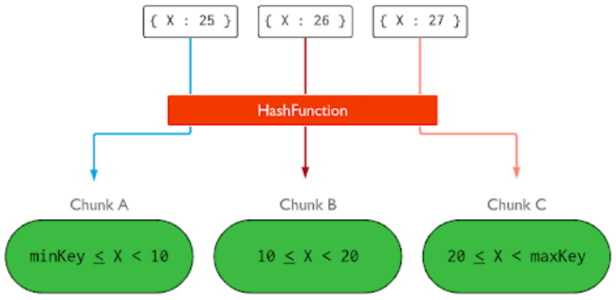


Figure 5: Hashed sharding [2].

2.2.2 Replication in MongoDB

MongoDB uses replication to address the need for high availability of data. This means that the same data can be redundantly stored on more than one node. Availability is improved by replication because duplication of data allows access to the data even when a node containing the data has failed. Another advantage of replication is that the replicated data is available for reading on multiple nodes, allowing for the distribution of read operations on that data [12]. The user can set the "replication factor", which is the number of nodes the data should be duplicated to. The set of nodes that contain the same data due to replication is called a "replica set". In general, a replica set should contain at least 3 nodes.

MongoDB follows a master-slave model to replicate data. That is, one node in the replica-set is the master/primary node, while all other nodes in the replica set are slave/secondary nodes. The nodes in the replica-set confirm their health to each other using a mechanism of "heartbeats". If the master node fails at any point, an election is conducted among the slave nodes to choose the new primary. Figure 6 shows an architecture where a client writes to a replica set with 3 nodes.

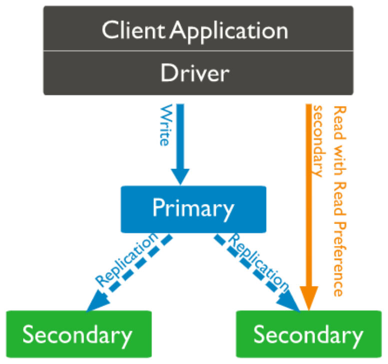


Figure 6: MongoDB replica set [2]

All writes to the replicated data are always directed to the master node. When writes are performed on the master node, a corresponding entry is created in the "oplog" of the master [12]. This oplog entry is used by the slave nodes to asynchronously apply these changes on their own copies of the data. In the case of reads, the user can specify the read preference by choosing between reading only from primary or preferring to read from primary or preferring to read from secondaries. Allowing to read from secondaries enables the distribution of reads over multiple nodes, but runs the risk of reading outdated data [12].

2.2.3 Monitoring Tools in MongoDB

While MongoDB does not provide a comprehensive monitoring system, MongoDB documentation [2] provides several command-line tools and utilities that can be used to view metrics that users may be interested in. These existing services save the metrics for 24 hours, after which they are discarded. Two useful utilities that can be used to obtain metrics from MongoDB are:

- mongostat: mongostat provides information about the instance that the utility is executed on. This includes query statistics, memory information, network traffic details, etc. mongostat can be run on a mongod or a mongos.

- mongotop: For each collection, mongotop shows details related to the time spent on reads and writes to that collection.

The following commands in MongoDB also report metrics that could contribute to the information monitored by a monitoring system:

- db.serverStatus(): This extremely useful command provides a wide variety of information about the instance on which it is run.
- dbStats or db.stats(): This command displays database-specific information such as the number and size of objects in the database.
- collStats or db.collection.stats(): This command displays collection-specific information similar to dbStats for databases.
- replSetGetStatus or rs.status(): When executed on a node in a replica-set, this command gives information about the replica-set, as well as node specific replication related information such as time heartbeats were sent/received to/from other nodes, election-related priority, etc.
- db.printShardingStatus() or sh.status(): Displays sharding related information. This included shard-key range split, chunk distribution, etc. It should be executed on a mongos.

2.3 Cassandra

Cassandra is a peer-to-peer, wide-column NoSQL database [13]. Similar to MongoDB, Cassandra is a distributed datastore. Apart from its features as a NoSQL datastore, it also provides a powerful query language that is extremely similar to SQL called CQL (Cassandra Query Language). The data in Cassandra are stored in key-spaces which could be equated to databases in relational database terms. Key-spaces contain tables that are similar to the concept of tables in relational databases and can be described as a collection of rows [3]. Rows are themselves a collection of columns and their values as key-value pairs. The diagram of Cassandra in Figure 7. shows a clear picture of the data model. The primary key for rows in Cassandra contains multiple fields: a partition key and an ordered set of clustering keys. Partition key acts similar to shard key as data is partitioned based on it. Clustering keys are optional and are mainly used for ordering data in a partition. Static columns are columns that are common to all entries with the same partition key, and need not be repeated for each entry.

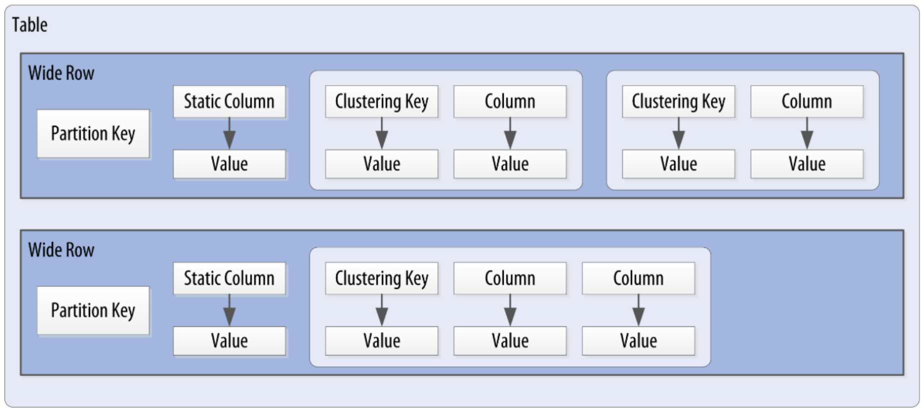


Figure 7: Cassandra Data Model [3].

The data written to Cassandra is partitioned across the nodes in the cluster for scalability. The partitioned data is also replicated to other nodes according to the replication factor. Unlike MongoDB, Cassandra follows a peer-to-peer replication model. That is, all nodes are equal in the role and there is no concept of master/primary nodes and slave/secondary nodes. The data is partitioned based on the hash value of the partition key of each row, known as Consistent Hashing. Based on the replication settings, the partitioned data is replicated across multiple nodes.

Read and write can be directed towards any node in the Cassandra cluster. This node then acts as a coordinator for that particular read or write. The coordinator node finds the nodes that contain copies of the data by applying the hash function of the key of the arriving write record and looking up its locations in a table. The value of “write consistency level” determines how many nodes the write should be replicated to before sending a success message back to the client and can be set by the user. To write data to a node, it is appended to a “commit log” for durability and is written to “memtable” which resides in memory. This makes writes to Cassandra fast. The contents in the memtable are occasionally flushed to “SSTables” which are “append-only” files residing on the disk. A similar path is followed to read from Cassandra, with the receiving node acting as a coordinator and directing the read request to the appropriate nodes by looking up where copies of the data are stored [3].

2.4 OpenTSDB

OpenTSDB is a time-series database built on top of HBase. Time-series databases are optimized to work on time-oriented data. Time-series data is increasingly relevant due to the widespread use of IoT devices that continuously generate records at fixed time intervals. Time is the most important attribute identifying each data-point in time-series data. For example, sensors in smart-watches and weather stations generate data that can be considered as time-series. Since monitored data is characterized by the time at which each data-point was collected, it intuitively seems that time-series databases may be a good choice for storing this type of data [14]. Hence, it is worth considering a popular time-series database like OpenTSDB as a candidate for the storage for the monitoring system.

HBase is a NoSQL database built on top of the Hadoop Distributed File System (HDFS), which allows large amounts of data to be stored in a distributed and yet durable fashion [15]. HDFS rose in popularity due to the advent of a new computing framework called MapReduce, which could parallel process tremendous amounts of data in a fast and durable way using HDFS as the storage [16]. The main advantage of HDFS, and in turn HBase, is that it is built with a level of abstraction that allows the user to interact with it as if it runs and stores data on a single machine. Therefore, even though OpenTSDB is built on top of HBase, users of OpenTSDB never have to interact directly with HBase to perform tasks in OpenTSDB. OpenTSDB stores its actual data in the underlying HBase database. OpenTSDB can be considered as an added layer above HBase to handle time-series data. Hbase has a master/slave architecture but OpenTSDB does not explicitly employ a master/slave architecture. Since the data that the user writes to OpenTSDB is being written to HBase, the partitioning of data across different nodes is handled by HBase and HDFS. Time-Series Daemon (TSD) is the background process that performs the core tasks for OpenTSDB. To perform any task in OpenTSDB, users communicate with TSD using a user-interface (UI) or an application programming interface (API) [4]. For example, reads can be requested by the user to the TSD, which fetches the data from the underlying HBase and serves it to the user. For writes, users direct the requests to the TSD, which in turn directs them to the underlying HBase master. Figure 8 shows the architecture of OpenTSDB.

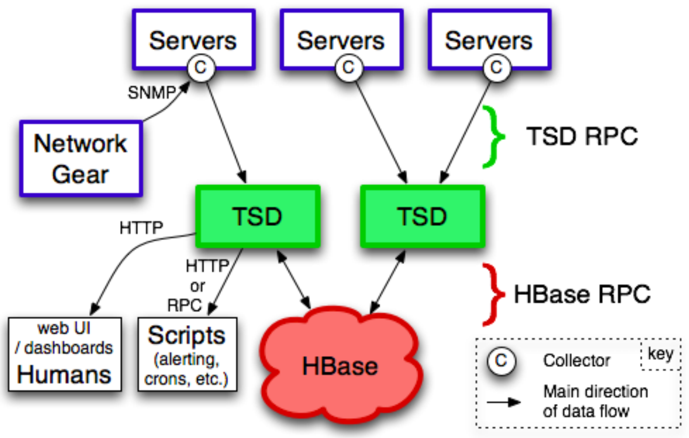


Figure 8: OpenTSDB Architecture [4]

Each data-point in OpenTSDB is identified with a combination of timestamp, metrics, and tags, and has a value associated with it. Metrics are the attributes of the monitored data being stored. For example, for time-series data that is generated by a weather station sensor, "temperature" and "pressure" could be two of the metrics. Tags are certain predefined fields that are used to identify data points. Time-series data stored in OpenTSDB should have one or more tags associated with them [4]. For the example of weather station data, possible tags are "weather station ID" and "sensor ID". That is, each record stored in OpenTSDB for this data would have a tag similar to "weather_station_id = x, sensor_id = y" associated with it.

CHAPTER 3

Related Works

The performance of NoSQL databases has been compared extensively in the literature, but there is no conclusive winner as the results depend on the type of data, architecture of the system and the modeling of the databases [17]. While monitoring systems for NoSQL databases haven't been the subject of much literature, systems for storing and monitoring similar data for other applications have been studied. In identifying relevant literature, the prime focus was to find papers on systems that received continuous streams of data from different sources at small and fixed intervals. These kinds of data are similar to the monitored data that is collected from each node in a MongoDB cluster at specified intervals. Since monitoring systems keep generating data continuously, often with the granularity of seconds, the velocity of the data is high. Also, depending on the number of devices or machines being monitored, a large number of data could arrive in the storage at every interval indicating a high volume of data. These characteristics of data point towards a need for big-data tools to handle monitoring data.

In [18], the authors implement a data repository for data generated by IoT devices. Specifically, this paper deals with RFID data and sensor data obtained from RFID tags that are used for tracking the tagged objects. MongoDB is considered as a candidate to store the data. The query performance of a single node MongoDB is compared with a relational database MySQL for the same queries, and it was found that MongoDB performs better than most queries even with only a single node. Also, a distribution test is performed on MongoDB by considering different shard keys and noting how the chunks are distributed across the cluster, with the aim being to select the shard key that provides the evenest distribution. The effect of scaling on MongoDB is also studied by conducting comparison tests related to volume and throughput on 1 node, 2 nodes and 3 nodes MongoDB. It was studied that volume and throughput are better as the number of nodes in the cluster increases.

Similarly, [6] proposes a system that monitors the radio spectrum to allow users to identify portions of the spectrum that are free or less crowded and can be reused. This system consists of the following components: The sensors that collect spectrum data, Kafka broker as a write buffer, a NoSQL database to store the collected data, a user interface between the users and the monitoring system, and a data processing engine to process data. Sensors collect the required data and contain software agents that perform some preliminary processing on the data. This data is then written to a Kafka broker which acts as a write buffer. MongoDB and Cassandra are considered candidates to store the data. They read the data in the Kafka broker and store it. The query performances for MongoDB and Cassandra were compared for several application-specific queries, and it was concluded that Cassandra is better suited for this use-case as compared to MongoDB. The performance advantage of using Kafka BigQueue is not experimentally studied in this paper.

In [5], cloud monitoring is performed to enable cloud systems to manage themselves (also known as autonomic cloud management systems). Metrics are collected and can be analyzed to decide when certain actions need to be taken. As in the cases of the sensor and IoT data, metrics from cloud servers arrive from different machines at a small interval. Smaller the interval, better the accuracy of the data. Therefore, the storage management of these metrics is modeled as a big data problem. In this paper, the authors have chosen to compare to wide-column NoSQL databases Apache Accumulo and Apache HBase to identify which has a better insert performance. The architecture of the monitoring system contains the following components: Monitoring plugins and plugin managers to obtain and process different types of monitoring data, data management layer which includes the BigQueue and is responsible for providing input to the application, and a data storage layer which is responsible for storing the monitoring data. The write operation times for Accumulo and HBase are compared, and it was concluded that Accumulo provides a better operation time as compared to HBase. This is attributed to the fact that Accumulo has multiple masters which allows it to distribute writes. Also, Accumulo provides write buffers for each client. Similar to [6], this paper also studies the effect of adding a write buffer known as "BigQueue" on write throughput. Instead of writing the data directly to HBase and Accumulo, the data is written to BigQueue. The data from BigQueue is written in batches to the HBase/Accumulo. This eliminates the extra time taken to establish connections before writing to the databases. It was noted that BigQueue drastically improves the performance of HBase by a factor of 300, and also improves the performance of Accumulo by a factor of 5. Accumulo is less affected by BigQueue as compared to HBase as Accumulo already has write buffers for each client.

CHAPTER 4

BigQueue

While surveying existing monitoring systems, it was noted that two of the implementations incorporated a write buffer into their architecture: [6] used a Kafka broker and [5] utilized a Java BigQueue for this purpose. Since the type of data and some of the architecture of these monitoring systems are comparable to that of the MongoDB monitoring system being built in this paper, it is worth considering if adding a write buffer would improve the write performance of our monitoring system. [5] compares the performance of HBase and Accumulo as the storage for their monitoring system with and without BigQueue and concludes that BigQueue provides a write performance improvement between 5 times and 300 times. It appears that using a BigQueue in the implementation of this project could provide substantial improvements in write performance. Therefore, we aim to study and compare the effects of BigQueue on the storage candidates of our monitoring system. The architecture of Big Queue represented in [5] is shown in Figure 9.

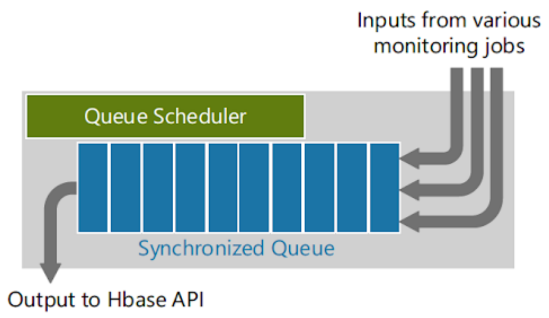


Figure 9: Architecture of Big Queue [5].

Big Queue [19] is a general concept used to describe a queue that is persistent, fast and follows the concepts of a Producer-Consumer [20]: several producers may concurrently write messages to the queue and several consumers may concurrently read messages from the queue. There are several standard implementations of Big

Queue. Java programming language has a Big Queue library that is commonly used as a persistent queue [21]. The queue is stored as a file in the file-system [19]. Similar implementations exist in other languages such as Golang.

One popular example of a BigQueue is Kafka, which is a messaging system developed at LinkedIn, primarily for log processing [22]. It follows a Producer-Consumer architecture where Producers send data to the Kafka Broker, which stores it to the file-system for Consumers to read. Messages coming into Kafka are classified as "topics" and Consumers can subscribe to certain topics. The messages written in Kafka are persistent. That is, durability is ensured for the data for the specified retention period. Kafka is designed to work across a cluster of nodes and fits very well with big data. The messages written to Kafka are also replicated across nodes for better durability [23]. The architecture of Kafka is shown in Figure 10.

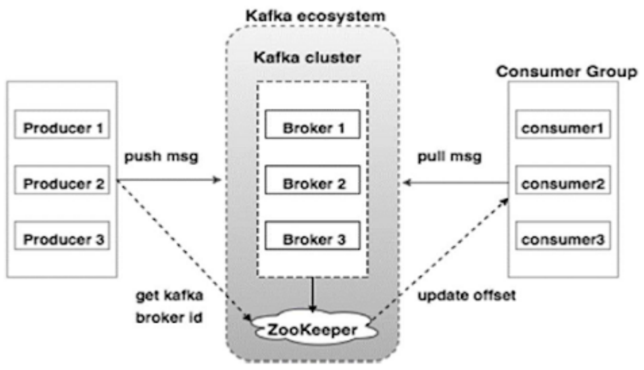


Figure 10: Kafka architecture.

Amazon provides a service called Kinesis [24] that is very similar to Kafka. For Kafka, the user has to bring up clusters and manage them, requiring significantly more effort. Amazon Kinesis is a better alternative for basic stream data processing because it is provided as an Amazon-managed service where the users simply have to configure it to their needs. Kinesis is less flexible than Kafka but takes significantly lower effort and time to configure and use. Kinesis also provides the advantages of Kafka, such as scalability and durability. In this project, the expectations from BigQueue are fairly simple: the ability to quickly write to the buffer and read from it in batches. For this purpose, there is no additional value to be gained from using Kafka instead of Kinesis. Using Kinesis, however, provides the advantage of the ease of use. Therefore, Kinesis is used as BigQueue in this project.

CHAPTER 5
Design and Implementation

To implement the proposed monitoring system, several components have to be built and combined together. The components necessary to build the monitoring system and run the experiments are:

1. The MongoDB cluster to be monitored.
2. Monitoring agents that run on each node of the MongoDB cluster and obtain relevant data from them.
3. Storage Candidates (NoSQL databases) to store the data collected by the monitoring agents.
4. A write buffer or BigQueue to act as a buffer between the monitoring agent and datastore.

5.1 Architecture

The MongoDB cluster to be monitored by the monitoring system (referred to as the Monitored MongoDB Cluster in the rest of the paper) should be up and running so that the monitoring system can obtain monitoring data. This cluster should have multiple nodes with replication and sharding enabled to represent real-world scenarios. A YCSB (Yahoo! Cloud Serving Benchmark) [25] client issues workload to this cluster to simulate a real database usage scenario. To obtain metric data from the Monitored MongoDB Cluster, a Monitoring Agent should run on each node in the cluster. The data obtained from the monitoring agent is to be stored in a NoSQL database (referred to as Storage Candidate in the rest of the paper to differentiate it from Monitored MongoDB Cluster). There are three Storage Candidates in this project: MongoDB, Cassandra, and OpenTSDB. Since the impact of using a write-buffer is to be analyzed, experiments are conducted with and without an Amazon Kinesis write buffer. In case a buffer is not used, monitoring agents directly write data to the NoSQL datastores as in Figure 11. In case a write-buffer is used, the agents should write data to the buffer which is then written to the databases in batches as in Figure 12.

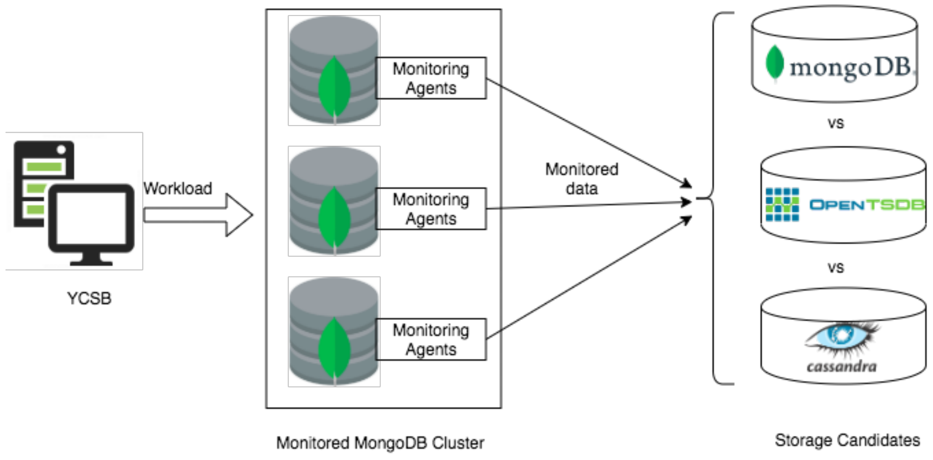


Figure 11: Architecture without BigQueue.

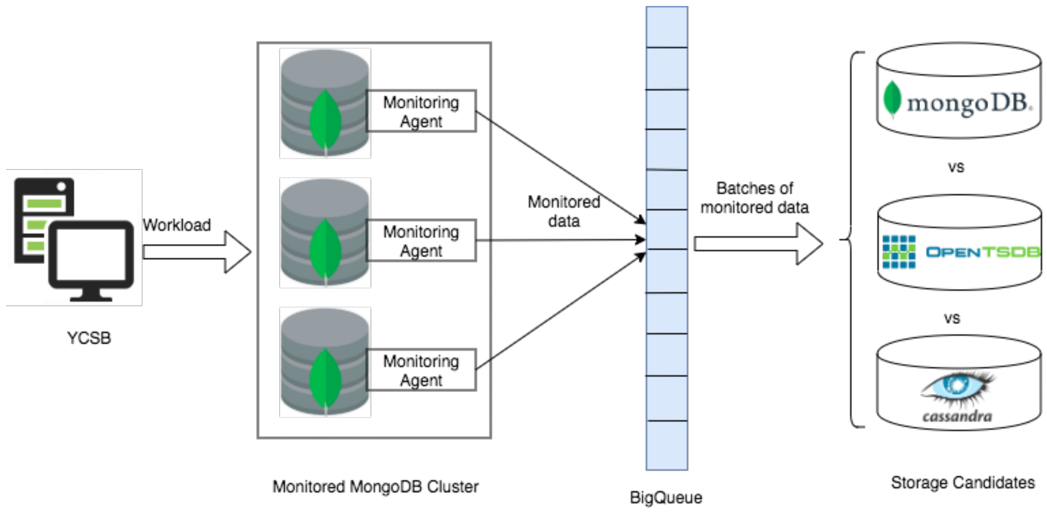


Figure 12: Architecture with BigQueue.

5.2 Monitoring Agents

Monitoring Agents are implemented as Python scripts that can obtain relevant information from the node that is run on. Different monitoring agents are implemented for MongoDB Storage Candidate, Cassandra Storage Candidate, and OpenTSDB Storage Candidate. This is because the data is modeled and written in very different ways in these databases. However, the same monitoring data are stored in each Storage Candidate under test.

In Section 2.3.3, several relevant commands, tools, and utilities provided by MongoDB have been identified. However, these commands and utilities contain a lot of information that a user of a monitoring system might not deem relevant. Also, they could be missing important information that a user might expect. To identify what data is to be monitored, a detailed survey of the metrics provided by popular NoSQL databases like CouchBase, HBase, Redis, and DynamoDB was conducted. From this survey, the most relevant monitoring information was identified and grouped into the following classes: instance details, query details, sharding details, and replication details. It was also identified that MongoDB does not provide a way to directly access the cache miss ratio. Sections 3.4.1 to 3.4.4 describe the categories of monitoring information that will be monitored from MongoDB, The commands/utilities that provide this information and the fields that are included. Section 3.4.5 describes how the cache miss ratio can be calculated in MongoDB.

5.2.1 Instance/Node details

This category contains fields that contain information about the instances and processes in the Monitored MongoDB Cluster. This also includes information related to network traffic. Table 1 summarizes the fields relevant to this category, the utility/tool containing them and their description.

Instance/Node Details		
Field name/ Command	Utility	Description
top		Shows all current processes; can be used to find CPU utilization
command	mongostat	No.of commands (per second)
vsize	mongostat	Virtual memory used by process (mongod/-mongos)
res	mongostat	Resident memory used by process(mongod/mongos)
qr	mongostat	No.of clients waiting in read queue
qw	mongostat	No.of clients waiting in write queue
ar	mongostat	No.of clients reading
aw	mongostat	No.of clients writing
netIn	mongostat	N/w traffic received by the instance
netOut	mongostat	N/w traffic sent by the instance
conn	mongostat	Total no.of connections that are currently open
process	serverStatus	The MongoDB process (mongod or mongos)
uptime	serverStatus	No.of seconds the process has been running
localTime	serverStatus	Current UTC time

Table 1: Monitoring Agents - Instance/Node Details

5.2.2 Query details

This category includes information such as the query statistics and log information. Query statistics show the number of queries of each type (insert, read, update, delete). The log profiling feature of MongoDB enables users to log queries slower than a given threshold. These query logs can, in turn, be queried like any other collection. Table 2 summarizes the fields relevant to this category, the utility/tool containing them and their description.

Query Details		
Field name/ Command	Utility	Description
insert	mongostat	No.of inserts
query	mongostat	No.of queries/reads
update	mongostat	No.of updates
delete	mongostat	No.of deletes
db.system.profile.find().limit(n).sort({ts:-1})	profiling	To return the most recent n log entries
db.system.profile.find(millis:{$gt:n})	profiling	Show all queries slower than n ms

Table 2: Monitoring Agents - Query Details

Sharding details		
Field name/ Command	Utility	Description
shards	db.printShardingStatus()	All the shards in the cluster and their instance/host, number of chunks in each shard, range of documents (with respect to shard key) in each shard.
explain()		To find which shard served the query
config.changelog	config db	Can identify Hotspots by looking at number of splits

Table 3: Monitoring Agents - Sharding Details

5.2.3 Sharding details

This category enables the user to view sharding details. We can also identify which shard served each query with the explain() function. Another useful information is the number of splits that are occurring in each instance. A large number of splits on an instance is a good indicator of hotspots. Table 3 summarizes the fields relevant to this category, the utility/tool containing them and their description.

Replication details		
Field name/ Command	Utility	Description
db.adminCommand({ replSetGetStatus :1})	replSetGetStatus	Returns the replica set status when run in the admin database.
heatbeatIntervalMillis	replSetGetStatus	Heartbeat frequency
members	replSetGetStatus	Details of each member in replica set (name, role, health)
set	mongostat	Name of replica set
repl	mongostat	M - master, S- secondary, REC-recovering, ARB - arbiter, UNK- unknown

Table 4: Monitoring Agents - Replication Details

5.2.4 Replication details

It contains details of replication. When the respective commands are executed on an instance, they return the status of the replica set from the executing server's perspective.

5.2.5 Calculating cache miss ratio

Cache miss ratio refers to the proportion of the memory accesses that resulted in a cache miss. Cache miss rate is a useful metric for users. However, it is not directly provided by any of the utilities or commands in MongoDB. It is possible to calculate the cache miss rate indirectly from some of the other metrics that MongoDB does provide. db.serverStatus() contains storage engine specific information. Among these fields, the following fields help us calculate cache miss rate [2]:

- Pages requested from the cache
- Pages read into the cache

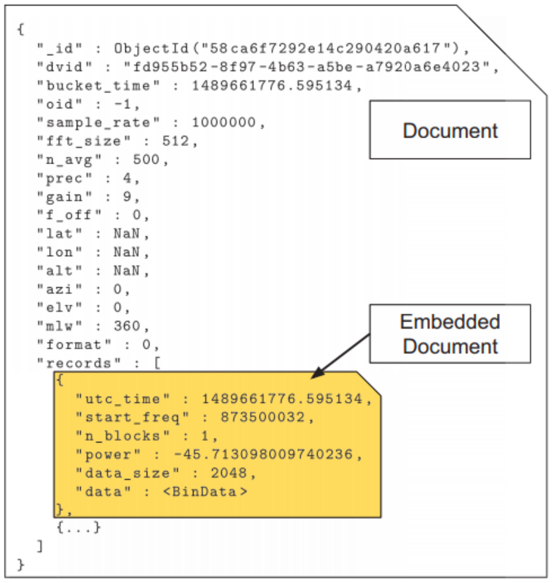
Since "Pages read into the cache" refer to the number of times a page that was being read was not present in the cache and had to be read into cache, it helps us calculate cache miss ratio using the following formula:

Cache miss ratio = (Pages read into cache/Pages requested from the cache)

5.3 Storage Candidates

This section presents the modeling of the Storage Candidates (MongoDB, Cassandra, and OpenTSDB) to store the data monitored from the Monitored MongoDB Cluster. Since each database under consideration has different data models, the best modeling for one database might not be the best for the other. The focus of modeling in this project is to offer a fair comparison by modeling each of the databases in a way that is best suited for them.

In the case of monitoring systems, records are being emitted continuously at very small intervals from several monitoring agents such as sensors. For example, in the case of our monitoring system, if the Monitored MongoDB cluster has three nodes and data is collected every second (monitoring interval = 1 second) from each node by the monitoring agent, 3*60*60 = 10,800 records are generated each hour. Since many real-life clusters to be monitored tend to have hundreds of nodes, the number of records generated will be extremely high. If we model the MongoDB Storage Candidate to write one document for each record being created, the number of documents stored will grow drastically, affecting the performance of the database. MongoDB recommends that instead of storing one document per monitoring interval, a technique called "bucketing" can be used [26]. In bucketing, multiple records can be embedded into another document which is characterized by a "bucket time". For example if bucketing is to be done per minute, all records with timestamps from (dd/mm/yy:hour:min:00) to (dd/mm/yy:hour:min:60) that are generated in the same minute "min" are embedded into a single document with the bucket time = (dd/mm/yy:hour:min). This technique was also adopted in [6], as shown in Figure 13. The concept of bucketing is used to model the MongoDB Storage Candidate in the implementation of this project. However, care should be taken not to put too much data into the same MongoDB document, as MongoDB restricts the maximum possible size of documents to 16MB [27]. Records from the same host and process with the same bucket times are embedded into a document. This is done such that ten monitored records are nested into one MongoDB document.



```
{
  "_id" : ObjectId("58ca6f7292e14c290420a617"),
  "dvid" : "fd955b52-8f97-4b63-a5be-a7920a6e4023",
  "bucket_time" : 1489661776.595134,
  "oid" : -1,
  "sample_rate" : 1000000,
  "fft_size" : 512,
  "n_avg" : 500,
  "prec" : 4,
  "gain" : 9,
  "f_off" : 0,
  "lat" : NaN,
  "lon" : NaN,
  "alt" : NaN,
  "azi" : 0,
  "elv" : 0,
  "mlw" : 360,
  "format" : 0,
  "records" : [
    {
      "utc_time" : 1489661776.595134,
      "start_freq" : 873500032,
      "n_blocks" : 1,
      "power" : -45.713098009740236,
      "data_size" : 2048,
      "data" : <BinData>
    },
    {...}
  ]
}
```

Figure 13: Example of Bucketing MongoDB Documents [6].

Another important factor to consider to model MongoDB Storage Candidate for storing monitored data is the selection of shard-key. While it is clear that time, host and process are important fields to identify each record, it is not immediately clear which single or composite shard-key combination offers the best distribution. In order to identify the best shard-key, various combinations of the host, process, and time fields will be considered in the experiments.

In Cassandra, the schema is defined when the table is created using a CQL query.

The primary key can be specified in the CQL query as a tuple of fields. The first field in the tuple acts as the partition key, while the other fields act as clustering keys in the given order. After surveying similar monitoring systems with Cassandra as the storage, Cassandra in our project is modeled by setting a primary key consisting of host, process, and timestamp. Host acts as the partition key. This ensures that records with the same host are stored in the same partition, in the order of process followed by timestamp. Process and timestamp act as clustering keys. In [28], the primary key of Cassandra that is used to store weather station data is defined in a similar way. That is, weather station ID is set as the partition key, while timestamp and some other weather-specific fields are set as clustering keys

Since OpenTSDB is specifically designed to store time-series data, time is implicitly used to identify each data point. Therefore timestamp is mandatory and separate from other fields in the monitored data. Host and process are also used to identify monitored records. Therefore "host" and "process" fields are set as the tags while modeling OpenTSDB Storage Candidate. For example, records from host A and mongos process are tagged with "host = A, process = mongos". Every other field in the monitored data is set as a metric in OpenTSDB. The partitioning of data across nodes is implicitly handled in the underlying HBase and HDFS.

CHAPTER 6

Experiments and Results Analysis

The experiments conducted in this project are classified into three categories: shard key test, Storage Candidate performance tests, and BigQueue test. In the shard key test, the best shard-key is identified by studying the distribution of objects across shards. This shard-key is then used in the sharded MongoDB Storage Candidate cluster for the remainder of the experiments. In Storage Candidate performance tests, the three Storage Candidates are compared with respect to read performance, write performance and disk storage requirements. These are used to evaluate the best Storage Candidate. Finally, the BigQueue test compares the writes performance with and without a Kinesis write buffer for each Storage Candidate. This helps in studying the effects of a write buffer and the extent of the impact on each Storage Candidate.

6.1 Experiment Set-up

In order to perform the experiments, four clusters of three machines each are set up on Amazon Web Services(AWS). The details of these clusters are given in Table 5.

Cluster	Service	No.of Nodes	Instance type
Monitored MongoDB	EC2	3 (1 master and 2 slaves)	m5.xlarge
MongoDB Storage Candidate	EC2	3 (1 master and 2 slaves)	m5.xlarge
Cassandra Storage Candidate	EC2	3	m5.xlarge
OpenTSDB Storage Candidate	EMR	3	m5.xlarge

Table 5: Details of clusters set up on AWS

Apart from these twelve machines, a separate EC2 machine is brought up to act as a client to the Monitored MongoDB Cluster. YCSB (Yahoo! Cloud Serving Benchmark) [25] is run on this machine to generate workloads containing a mix of reads, inserts, updates and deletes to the Monitored MongoDB Cluster. This helps in simulating the working of a real database. Once each cluster was set-up and monitoring agents were implemented separately for each of the three Storage Candidates, the monitoring was performed on the Monitored MongoDB Cluster for exactly one week. At the end of one week, each Storage Candidate (MongoDB, Cassandra, and OpenTSDB) will contain the monitored data for that period. This is done to prevent the experiments from being affected by the biases of being performed on empty Storage Candidates. BigQueue for the experiments is implemented with Amazon Kinesis.

6.2 Shard-Key Test

For Cassandra, the partitioning of data across nodes is based on the partition key. OpenTSDB data partitioning is handled by the underlying HBase and is not under the user's control. However, the shard-key for MongoDB has to be determined explicitly by the user. This test attempts to find the best shard-key for MongoDB Storage Candidate. To identify the best shard key, the candidate shard keys are first identified. The fields of relevance are timestamp, host (the particular machine in the Monitored MongoDB Cluster) and process (mongod or mongos). Each of these fields and their combinations are considered as candidate shard-keys. The effects of the candidate shard keys are studied on MongoDB Storage Candidate, and the best shard-key is identified. To identify the best shard-key, sharding is performed on the same data with each candidate shard-key. This is done by copying the data into new collections with different shard-keys. After sharding, the distribution of documents across shards and the number of chunks are used to identify the best shard-key. The aim is to choose the shard-key that provides a reasonable number of chunks and even distribution of documents. The choice of shard-key affects how data is distributed across nodes, as all documents with the same shard-key go to the same chunk. The chunks are then distributed across nodes. A shard-key with low granularity would result in a low number of chunks, which cannot be further split. It is important to select a shard-key that allows MongoDB to split chunks when they grow too big [29].

6.2.1 Results

The results of the Shard-Key test are summarized in Table 6. Every shard starts out with two empty chunks. When data is populated in these shards, they go to these chunks. When chunks grow big, they split and form new chunks. Therefore, at any point, there are at least two chunks in each shard, one or more of which could be empty if they haven't been populated with data at all. Empty chunks are not considered in Table 6 as they do not add value to our findings. An example where empty chunks are ignored in Table 6 is when "Process" is used as shard-key. Since process could only have two possible values ("mongos" or "mongod"), two unsplittable chunks are formed, both of them in Shard 1. Shard 2 and Shard 3 contain two empty chunks each. These are ignored, and Shard 2 and Shard 3 are considered to contain not chunks.

Shard-key Candidate	No.of documents in Shard 1	No.of documents in Shard 2	No.of documents in Shard 3	No.of chunks in Shard 1	No.of chunks in Shard 2	No.of chunks in Shard 3
Bucket_ts	479939	316456	398024	5	6	7
Host	0	409174	785245	0	1	2
Process	1194419	0	0	2	0	0
(Bucket_ts, Host)	368900	391901	433618	11	12	12
(Bucket_ts, Process)	371476	352027	470916	7	7	8
(Host,Process)	397143	410086	387190	1	2	1
(Bucket_ts, Host, Process)	398944	398113	397362	14	14	14

Table 6: Shard-key Test Results

6.2.2 Analysis

The following observations can be made from the results of the Shard-Key Test:

- *Observation 1:* Low granularity shard-keys and their combinations, such as "Host" (granularity of 3), "Process" (granularity of 2) and "Host, Process" (granularity of 4) cause unsplittable chunks and poor distribution of documents across shards. Therefore, they are bad shard-key candidates. (Note: "Host, Process" has a granularity of 4 as the distinct values possible are "A, mongod", "B, mongod", "C, mongod", and "A, mongos", where A, B, and C are the three nodes in Monitored MongoDB Cluster.)
- *Observation 2:* When "Bucket_ts" is added to low granularity keys, it's high granularity provides the ability to split the chunks when they grow too large [29]. "Bucket_ts, Host, Process" as the shard-key gives the best distribution of chunks and documents across shards.

Conclusion: (Bucket_ts, Host, Process) is set as the shard-key for MongoDB Storage Candidate.

6.3 Storage Candidate Disk Storage Test

Once the Storage Candidates are populated by monitored data collected over one week, disk storage can be measured directly for each Storage Candidate and compared. The size obtained will be dependent on the compression algorithms used by the database and the way in which data is stored in the database.

6.3.1 Results

The disk storage space depends on the way data is stored in each Storage Candidate and the compression algorithms used by them. The disk storage spaces used by the Storage Candidates to store monitoring data collected over the same time period (of a week) are compared in Figure 14.

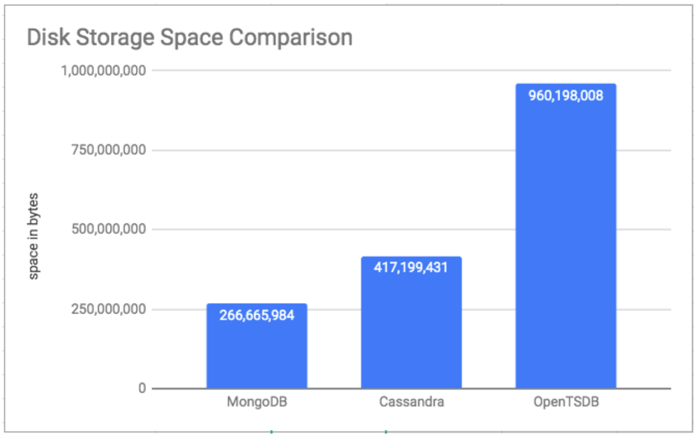


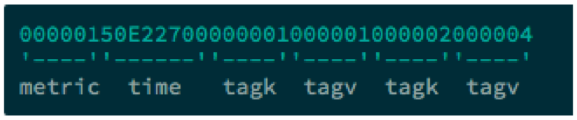
Figure 14: Disk Storage Results.

6.3.2 Analysis

The following observations can be made from the results of the Storage Candidate Disk Storage Test:

- *Observation:* OpenTSDB has the worst disk storage space among all the Storage Candidates. This can be attributed to the way the data of OpenTSDB is stored in HBase. As shown in Figure 15, each metric for each time-point and each tag combination is stored in a different row in HBase. On the other hand, Cassandra and MongoDB combine metrics from the same timestamp or bucket time, process and host into the same rows and documents respectively. The number of rows in HBase on behalf of OpenTSDB is therefore extremely high compared to the number of rows in Cassandra and the number of documents in MongoDB. MongoDB shows the best disk storage, which in part can be because of bucketing, where ten documents within a time-range are embedded into a single document.

Conclusion: MongoDB (with bucketing) provides the best disk storage compared to Cassandra and OpenTSDB for data obtained in this monitoring system.



```
00000150E22700000001000001000002000004
'----''------''----''----''----''----'
metric  time    tagk  tagv  tagk  tagv
```

Figure 15: Underlying HBase Row-key for OpenTSDB Data [4].

6.4 Storage Candidate Query Test

Four different types of read queries are identified. The queries are defined such that they are the most common types of queries expected by our Monitoring System. Each query is executed on each Storage Candidate ten times, and the average execution time is calculated. The performance comparison between Storage Candidates is performed separately for each query. Query 1 through Query 3 fetches records with increasing restrictiveness. Query 4 is an aggregate query. The function of the queries are described below:

- Query 1: All documents between time A and B across all hosts.
- Query 2: All documents between time A and B in one specific host.
- Query 3: All documents matching given time range, host and process.
- Query 4: Sum of the network traffic input over a given day.

6.4.1 Results

The comparison average times for Query 1, Query 2, Query 3 and Query 4 for each Storage Candidate are shown in Figure 16, Figure 17, Figure 18 and Figure 19 respectively.

6.4.2 Analysis

The following observations can be made from the results of the Storage Candidate Query Test:

- *Observation 1:* Cassandra starts with the worst performance, but improves in performance as the queries become more restrictive. The performance of Cassandra for Query 1 and Query 2, Cassandra performs much worse than

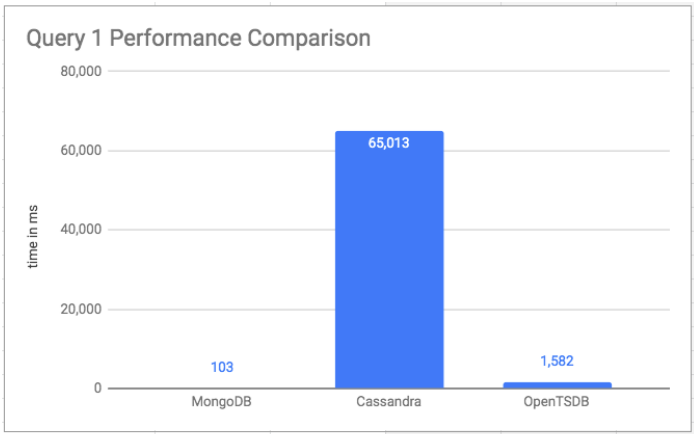


Figure 16: Comparison of Performance on Query 1.

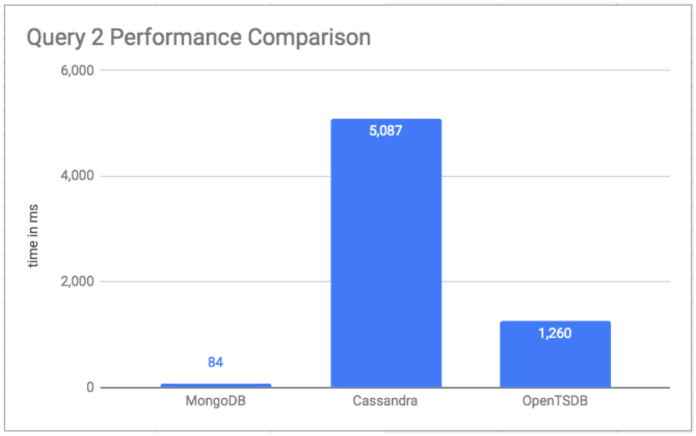


Figure 17: Comparison of Performance on Query 2.

MongoDB and OpenTSDB. The time taken by Cassandra for Query 1 and Query 2 are because all fields of the primary key are not restricted in these queries. In Query 1, only the timestamp is provided. Since "host" is the partition key, Cassandra in Query 1 doesn't get enough information from the query to restrict it's search to certain partitions, and end up having to search in all partitions. In Query 2, "host" is specified along with timestamp which improves the performance by nearly 13 times. In Query 3, all primary key fields are restricted. In general Cassandra is considered to be a write optimized database, and not read optimal.

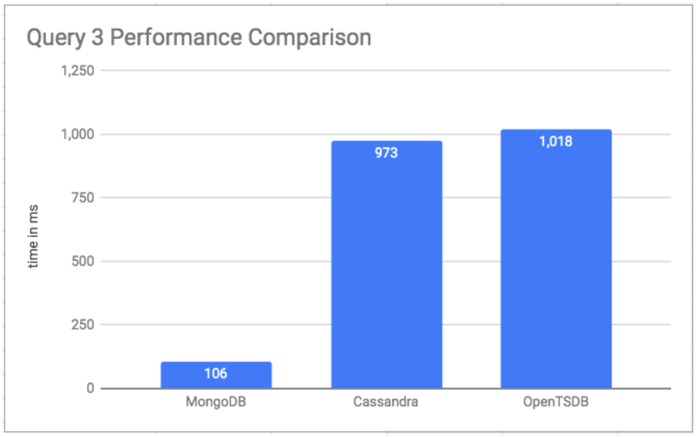


Figure 18: Comparison of Performance on Query 3.

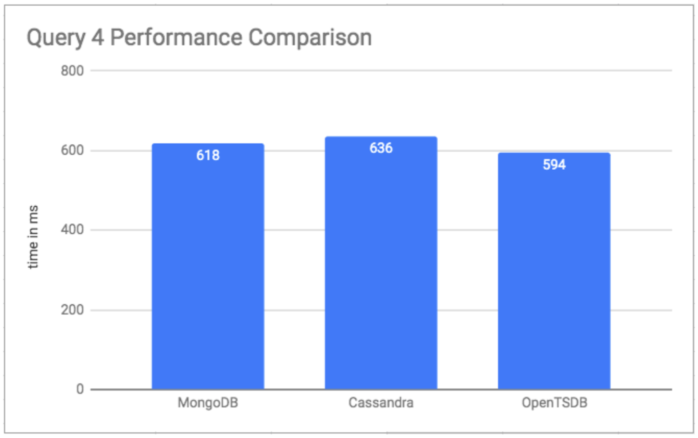


Figure 19: Comparison of Performance on Query 4.

- *Observation 2:* OpenTSDB performs worse than MongoDB for most queries. Most of this difference can be explained by two reasons. Firstly, bucketing reduces the number of MongoDB documents that have to be examined and returned, reducing the time for the query to return. Secondly, the data in OpenTSDB is stored in the underlying HBase in such a way that each metric measured at each point in time is persisted in a different row. Therefore the number of rows in HBase will be much much higher than the number of documents in MongoDB even without bucketing. The data in HBase is ordered by metrics first, followed by timestamp and tags.

Conclusion: MongoDB with bucketing provides the best read performance considering all four queries.

6.5 Storage Candidate Write Test

Write performance is measured by the time taken to write 50000 and 500000 monitored objects as fast as possible into each Storage Candidate. Since MongoDB Storage Candidate is modeled by bucketing ten records into one document, 5000 and 50000 documents are written for the write performance test instead of 50000 and 500000 respectively. Apart from comparing the write times between Storage Candidates, this will also provide an insight to how the write time for each Storage Candidate is affected by increasing the number of monitored records.

6.5.1 Results

The comparison of the time in seconds taken to write 50,000 records (5000 buckets in case of MongoDB) to each Storage Candidate is shown in Figure 20. A similar comparison for 500,000 records is shown in Figure 21. For a clearer representation, the values are also depicted in Table. 7

Storage Candidate	Time for 50,000 writes (in seconds)	Time for 500,000 writes (in seconds)
MongoDB	5.588	50.140
Cassandra	2.5	20
OpenTSDB	297	3212

Table 7: Write Performance Results

6.5.2 Analysis

The following observations can be made from the results of the Storage Candidate Write Test:

- *Observation 1:* For MongoDB and Cassandra, the time taken to write 500,000 records is slightly less than 10 times the time taken for 50,000 writes. For

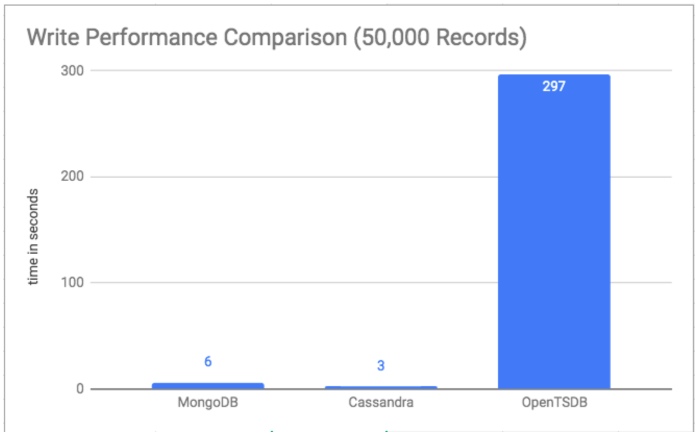


Figure 20: Comparison of time to write 50,000 Records.

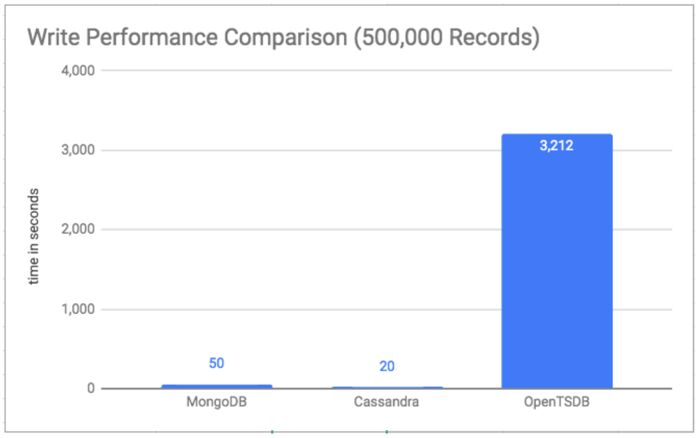


Figure 21: Comparison of time to write 500,000 Records.

OpenTSDB, time for 500,000 writes is slightly higher than the 10 times the time for 50,000 writes. This demonstrates that the write time with increases almost proportionally with the number of records written for all three Storage Candidates, but OpenTSDB write performance scales slightly worse in comparison.

- *Observation 2:* Cassandra performs fastest for writes. The exceptional performance of Cassandra for writes can be attributed to the way writes operations are performed in Cassandra. Cassandra follows a peer-to-peer architecture where writes can be directed to any node, eliminating the risk of bottlenecks at master nodes. Also, Cassandra considers writes are completed when it is written to the commit log and memtable. The memtable is occasionally flushed to disk, reducing the overall number of I/O operations required for writes.

- *Observation 3:* OpenTSDB exhibits the worst write performance. While the performance of MongoDB and Cassandra are comparable, OpenTSDB performs much worse compared to both. On the other hand, OpenTSDB is internally structured in a way that provides a disadvantage to write operations. The writes are directed towards TSD (Time Series Daemon), which may act as bottlenecks themselves. Once write requests are received by the TSD, they have to be forwarded to the underlying HBase master which is another potential bottleneck. Further, the data is stored in the underlying HBase such that each metric that is coming in at each timestamp and for each distinct tags is persisted in a different row in HBase, as shown in Figure 15. The number of rows to be written in HBase is therefore much larger than that in Cassandra, contributing to the much longer time taken for writes in OpenTSDB.
- *Observation 3:* MongoDB performs well in write performance, and only slightly worse than Cassandra. In part, this can be attributed to bucketing. Since ten records are embedded into one document, we are effectively only writing one document for every ten records.

Conclusion: Cassandra provides the best write performance for both 50,000 and 500,000 records.

6.6 BigQueue Test

In this test, we aim to investigate if introducing batch ingestion with the help of a Kinesis write buffer will improve write performance of the Storage Candidates. Instead of writing each monitored record into the Storage Candidate one by one, this allows successive writes to be stored in the Kinesis queue and then written to the Storage Candidates in a single batch as one write. As discussed in Chapter 2, this is expected to provide the advantage of a reduction in overhead of establishing connections that are required for each write operation. The BigQueue test measures the time to write monitored records to each Storage Candidate by first writing to Kinesis and reading from Kinesis in batches. These results are then compared to the write performance without BigQueue, which is already measured in the Storage Candidate Write Test. Similar to the Storage Candidate Write Test, the time taken to write 50000 and 500000 records into each Storage Candidate is calculated for the BigQueue Test. Note that in the case of MongoDB, 5000 and 50000 records are written instead, to account for bucketing.

6.6.1 Results

The time taken to write 50,000 and 500,000 records with and without BigQueue to MongoDB, Cassandra and OpenTSDB Storage Candidates are compared in Figure 22 and Figure 23 respectively. The results are also summarized in Table 8.

	Time for 50,000 Records (seconds)		Time for 500,000 Records (seconds)	
Storage Candidate	Without BigQueue	With BigQueue	Without BigQueue	With BigQueue
MongoDB	5.588	4.328	50.140	39.5
Cassandra	2.5	10	20	105
OpenTSDB	297	21	3212	216

Table 8: BigQueue Experiment Results

6.6.2 Analysis

The following observations can be made from the results of the BigQueue Test:

- *Observation 1:* BigQueue worsens the write performance of Cassandra. The performance of a database with BigQueue depends hugely on how batch inserts

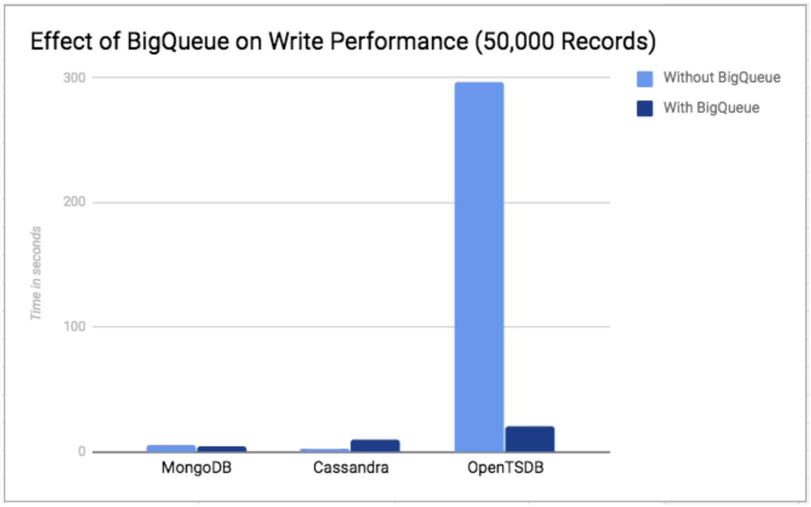


Figure 22: Effect of BigQueue on Write Performance (50,000 Records)

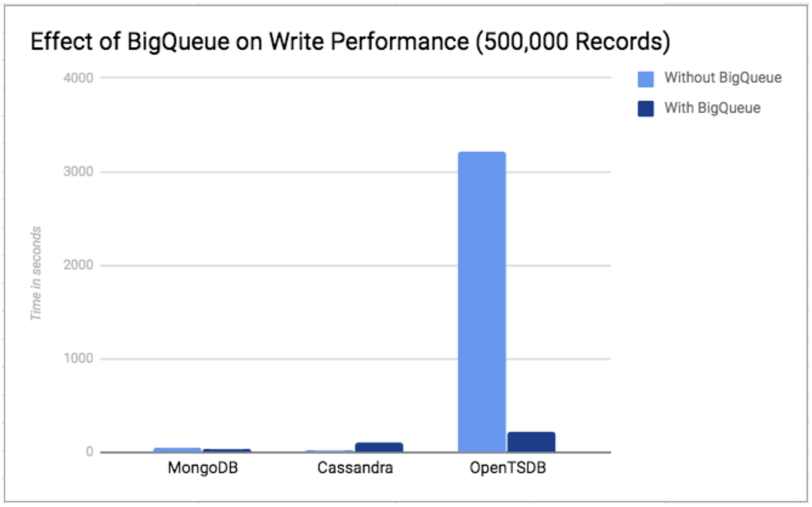


Figure 23: Effect of BigQueue on Write Performance (500,000 Records)

are supported in the database. In the case of Cassandra, it is not recommended to use batch inserts other than in cases where a group of operations has to be performed together. One instance where this would be relevant is to ensure the atomicity of transactions consisting of multiple operations: either all operations should be completed or none should be performed at all. Especially when multiple partitions may be involved, batch inserts worsen the write performance in Cassandra. This is because of the peer-to-peer architecture of Cassandra.

The batch writes may be directed to any node, which then has to act as the coordinator for each write by sending it to the relevant partition. Therefore, introducing BigQueue reduces the write performance of Cassandra, as BigQueue brings batch insertion into the picture.

- *Observation 2:* BigQueue slightly improves the write performance of MongoDB. Theoretically, we expect MongoDB to have a major improvement in write performance as the overhead to establish a connection with master while inserting one record at a time is eliminated. However, bucketing has already eliminated some of this overhead, which is reflected in the good write performance without BigQueue. Therefore the introduction of BigQueue does improve the write performance slightly but not drastically.
- *Observation 3:* BigQueue improves the write performance of OpenTSDB drastically. Without BigQueue, each write is directed to the TSD, which then communicates with HBase master to persist the writes. Thus there is significant overhead in the regular writes in OpenTSDB. The introduction of BigQueue along with batch inserts causes a large improvement due to the elimination of overhead.

Conclusion: BigQueue drastically improves OpenTSDB write performance, slightly improves MongoDB write performance and worsens Cassandra write performance.

CHAPTER 7

Conclusions and Future Work

In this project, a monitoring system was built to monitor a distributed MongoDB database. The most relevant information to monitor was identified and written to three Storage Candidates- MongoDB, Cassandra, and OpenTSDB. Experiments were conducted to find the best Storage Candidate for the monitoring system. Care was taken to ensure that comparison is fair by ensuring that each Storage Candidate was best modeled for our specific use-case.

It is observed that MongoDB with bucketing provides the best disk storage and read performance. As for write performance, Cassandra performs best but MongoDB closely follows. Using a BigQueue write buffer improves the write performance of MongoDB further, bringing its write performance even closer to that of Cassandra. Overall, it can be concluded that MongoDB with bucketing and BigQueue write buffer serves as the best Storage Candidate for the monitoring system proposed in this project.

In future studies, more NoSQL databases like InfluxDB, HBase, and Redis could be considered as Storage Candidates against MongoDB Storage Candidate. More criteria could also be introduced to evaluate them. Since the current focus of the project is to design an efficient monitoring system, a UI has not been implemented. In the future, a dashboard can be designed for this monitoring system so that users may be able to view summaries of monitored data.

LIST OF REFERENCES

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

[2] M. Inc., ‘‘MongoDB Manual,’’ https://docs.mongodb.com/manual/, 2019, [Online].

[3] E. Hewitt, *Cassandra: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[4] OpenTSDB, ‘‘OpenTSDB User Guide and Documentation,’’ http://opentsdb.net/docs/build/html/user_guide/, 2019, [Online].

[5] S. Zareian, M. Fokaefs, H. Khazaei, M. Litoiu, and X. Zhang, ‘‘A big data framework for cloud monitoring,’’ in *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, 2016, pp. 58--64.

[6] G. Baruffa, M. Femminella, M. Pergolesi, and G. Reali, ‘‘Comparison of mongodb and cassandra databases for supporting open-source platforms tailored to spectrum monitoring as-a-service,’’ *IEEE Transactions on Network and Service Management*, 2019.

[7] M. Stonebraker, ‘‘Sql databases v. nosql databases,’’ *Communications of the ACM*, vol. 53, no. 4, pp. 10--11, 2010.

[8] I. Amazon Web Services, ‘‘Db-engines ranking,’’ https://db-engines.com/en/ranking, 2020, [Online].

[9] P. Zikopoulos, C. Eaton, *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media, 2011.

[10] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Pearson Education, 2013.

[11] R. Copeland, *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database.* " O'Reilly Media, Inc.", 2013.

[12] K. Banker, *MongoDB in action.* Manning Publications Co., 2011.

[13] A. Lakshman and P. Malik, ‘‘Cassandra: structured storage system on a p2p network,’’ in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009, pp. 5--5.

[14] J. D. Hamilton, *Time series analysis.* Princeton New Jersey, 1994, vol. 2.

[15] L. George, *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107--113, 2008.

[17] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser, "Performance evaluation of nosql databases: a case study," in *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, 2015, pp. 5--10.

[18] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone, and V. N. Vitale, "Industrial internet of things: Persistence for time series with nosql databases," in *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE).* IEEE, 2019, pp. 340--345.

[19] "A big, fast and persistent queue based on memory mapped file," https://github.com/bulldog2011/bigqueue, 2013, [Online].

[20] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces.* Arpaci-Dusseau Books LLC, 2018.

[21] "Introduction to BigQueue," https://www.baeldung.com/java-big-queue, 2020, [Online].

[22] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1--7.

[23] A. S. Foundation, "Apache Kafka, A distributed Streaming Platform," https://kafka.apache.org/documentation/, 2017, [Online].

[24] I. Amazon Web Services, "Streaming Data Solutions on AWS with Amazon Kinesis," https://d0.awsstatic.com/whitepapers/whitepaper-streaming-data-solutions-on-aws-with-amazon-kinesis.pdf, 2017, [Online].

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143--154.

[26] R. Walters, "Time Series Data and MongoDB: Part 2 – Schema Design Best Practices," https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2-schema-design-best-practices, 2018, [Online].

[27] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in mongodb," in *2014 IEEE International Advance Computing Conference (IACC).* IEEE, 2014, pp. 416--421.

[28] D. Ramesh, A. Sinha, and S. Singh, ‘‘Data modelling for discrete time series data using cassandra and mongodb,’’ in *2016 3rd international conference on recent advances in information technology (RAIT)*. IEEE, 2016, pp. 598--601.

[29] W. Zola, ‘‘On Selecting a Shard Key for MongoDB - MongoDB Performance Best Practices,’’ https://www.mongodb.com/blog/post/on-selecting-a-shard-key-for-mongodb, 2015, [Online].