Data Storage Performance Collation

Apurva Jagdale
University of Passau
Passau, 94032, Germany
jagdal01@gw.uni-passau.de

ABSTRACT

"Data is the new oil!" [21] it is easy to perceive term as the way information (data) is used to shape the future using transformative technologies [40] we see today – artificial intelligence, automation and advanced, predictive analytics. Of course, data, like oil is a source of power. And those who control it (think of Amazon, Alibaba, Facebook or Google) are establishing themselves as masters of the universe. Data – particularly Big Data – has far more dynamic nature than oil. It can be attributed into 5 V's - Volume, Velocity, Variety, Value and Veracity. To store and retrieve this data, technologies which are normally based on distributed systems and parallel processing like NoSQL Databases comes into picture. Based on the nature, structure of the data along with the business use case, variety of NoSQL Database Technology Platforms are currently available. This paper compares the performance of storing and retrieving this data from popular NoSQL databases along with data streaming platform which resembles with database management system concept and also applies some common optimizations to improve the read/write performance of it. Experiment is specifically performed on MongoDB, Confluent Kafka, Cassandra and Elastic Search. Data model used is same across all the databases to avoid inaccuracies in the performance measurements.

KEYWORDS

NoSQL,Big Data,Streaming Data,Performance,Optimization

1 INTRODUCTION

Today applications in every domain generate or utilizes data which is dynamic and colossal in nature.NoSQL DBMS[**?**] have become popular due to their Multi-Model, Scalability, Flexibility, Distributed and the capability to manage large amounts of data. A huge amount of data requires systems which should be able to gear data writes as well as read efficiently.[7] In Real-Time Enterprise Applications data is a crucial factor; database systems which deliver good performance to fulfill the business need is required. This importance of Big Data applications has encouraged the development of a variety of NoSQL DBMS e.g. MongoDB, Facebook's Cassandra, Apache's Kafka, Confluent's Kafka, Elastic Search, Amazon's Dynamo.
From all present NoSQL storage solutions selection of database which fulfills the business requirements is necessary. This can be achieved using CAP [11] Theorem and the Data Model [12] which classifies NoSQL DBMS. But, after the selection of a suitable database, there is a need for doing configuration improvements, which will bring us to a database with the best read and write performance.
Purpose of this paper is to carry out the performance stress tests on popular NoSQL DBMS - MongoDB, Apache Cassandra, Elastic Search, Confluent Kafka. Here we have considered Confluent Kafka[32] which is not exactly a database but Streaming Platform. Also, MongoDB[37] as a document database, Elastic Search[36] which is document database search engine based on Apache Lucene library and Cassandra[31] which is a hybrid between a key-value and a tabular database management system for a case study. This will provide insights about the read and write performance of these distinct databases and will aid to choose a suitable one. Henceforth, we will refer to all storage platforms as a 'Database' for ease of understanding. Initially, each database is deployed on an individual VM with same hardware resource configuration with the same JSON schema populated in the databases. Python-Clients of the individual database used to perform reads and writes. Initially, all the performance readings are recorded without any configuration improvements. Secondly, after applying basic configuration and code optimization it leads to read/write performance improvement. Also, performed inbuilt stress tests which emulate big production setups on selective databases. Finally, collating all performance readings leading to the conclusion.
This paper is structured as follows. Section 2 discusses related work done for comparing different types of NoSQL databases. Section 3 and 4 lay foundations of the NoSQL databases by explaining Data-Model, CAP Theorem, and key features of Cassandra, Elastic Search, Kafka, MongoDB. Section 5 discusses about experimental setup. Section 6 is divided into 2 sub-sections Default and Optimized Configuration with experimental findings and analysis. Final two sections are the conclusion and future scope.

2 RELATED WORK

To date, there has been limited work in benchmarking the performance of NoSQL datastores. M. Jung et. al [14] compares the performance of NoSQL database against traditional RDBMS. The author uses PostgreSQL and MongoDB for his comparative analysis. By contrast, this paper benchmarks popular NoSQL databases for their read and write performance under the same virtual hardware configuration. In Gandini, Andrea et. al[7] author is focusing on three data-stores and explores their performance under different hardware, software, and workload configurations. Partly similar work has been carried out by Klein et. al [18] but differs in terms of a number of databases selected for the experiment. Also, this paper discusses the performance trade-off of data streaming platform which can be served as a database solution under certain conditions.

3 BACKGROUND

R. Hecht and S. Jablonski [12] has explained Data Model and Jing Han et. al[11] explains CAP Theorem (Consistency, Availability, Partition Tolerance) which are foundations for classification of the NoSQL database systems.
Data Model can be categorized as below.

3.1 Data-Model

"A database model is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized and manipulated. The most popular example of a database model is the relational model, which uses a table-based format." wikipedia[35]

3.1.1 Key-Value: A basic data structure used in this data storage paradigm is associative arrays or hash tables. Records are stored and retrieved using a key that uniquely identifies the record. Most graph databases are key-value database internally. e.g. Redis, Oracle NoSQL, and RocksDB

3.1.2 Column-oriented: Data is stored by column, that is data stored separately for each column. Each column of data is the index of the database. The advantage of this data model is a more suitable application on aggregation and data warehouse. e.g. MariaDB and Apache HBase.

3.1.3 Document-Based: Document database and Key-value is very similar in structure, but the Value of document database is semantic and stored in XML, YAML, JSON, and BSON, as well as binary forms like PDF. e.g. MongoDB

3.2 CAP Theorem

CAP theorem's core idea is a distributed system cannot meet the three district needs simultaneously, but can only meet two. This divides the databases into three different variants.[11] In case of database management systems third need is partially satisfied with provision of some configuration. further example can be seen in section 4.1.

3.2.1 consistency and availability(CA):. Use of replication for ensuring the data consistency and availability.system following CA are a traditional relational database and Vertica Column-oriented.

3.2.2 consistency and partition tolerance(CP):. Storing data in a distributed manner across the nodes but this kind does not support availability. e.g. BigTable (Column-oriented) and MongoDB (Document)

3.2.3 availability and partition tolerance (AP):. Such systems ensure availability and partition tolerance primarily. e.g.SimpleDB (Document-oriented) and CouchDB (Document-oriented)

4 DATA STORAGE SYSTEMS

In this section, we describe the background about different data storage frameworks that we examine in this paper.

4.1 Cassandra

Apache Cassandra is a hybrid between Key-Value store and Column-Oriented database which follows the availability and partition tolerance strategy from CAP theorem. In Figure 1. Data Model of the Cassandra is structured as, Column family with a way to store and organize data in different columns and table as a two-dimensional view of a multi-dimensional column family. Availability of data in Cassandra is assured by replication, replication factor can be set to replicate the data across the Cassandra cluster. Replication varying from one which is a single point of failure to a number of nodes in the cluster. When a keyspace is created with a 'SimpleStrategy' first replica is placed on the node determined by the partitioner. And replicas on the next nodes in the ring in a clockwise manner. There is 'NetworkTopologyStrategy' used when the cluster is deployed across multiple data centers. Replication and peer-to-peer model makes Cassandra fault tolerant storage system.[13]

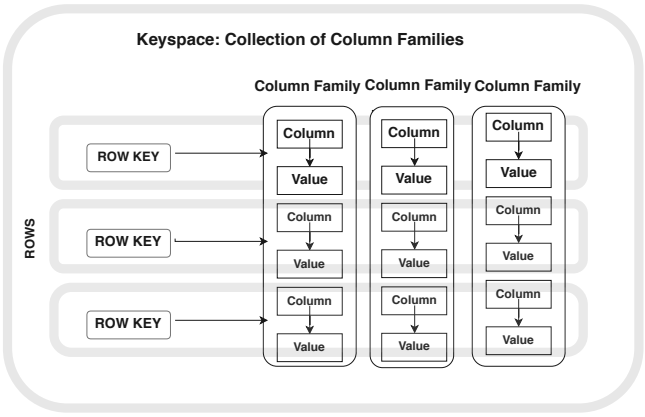

Figure 1: Cassandra Data Model [8]

Partitioning in Cassandra is done by partitioner and it helps to organize how the row keys should be sorted along with distribution of data across the Cassandra nodes.there are 'Random Partitioner' with MD5 hash applied to determine the place and 'Order Preserving Partitioner' with UTF8 string; data stored in the order of this key value in Cassandra.[13]

Consistency is handled using different consistency levels in Cassandra for read and write – zero, one, quorum, all. Consistency level affects the latency for the read and write operations. Therefore, fixing the consistency value for an application is important. Zero means write operations should be performed asynchronously and one means write request will return only when one node has written new data in commit log successfully after storing the key from write request.[13]

4.2 Elastic Search

Elastic Search[36] is an open source distributed real-time document store search engine built on top of Apache Lucene[33]. And it implements Availability and Partition Tolerance compromising Consistency.

Due to the indexing feature inherited from the Lucene, elastic search proves itself as a great search engine. Using elastic search you can search, filter, sort, index the data inserted. The largest unit in Elastic Search is index and the smallest unit is a document. An index can be seen as a collection of documents which contains data of common feature. In correlation with the RDBMS, 'Index' can be seen as a database schema. 'Document' stored inside an index can be seen as a Row and 'Mapping Type' which explains the common feature can be seen as a Table.

Further, each index can be divided into multiple shards, and elastic search exclusively takes responsibility of it.[5] when we dig inside the documents, we see multiple fields and based on these fields the index is assigned to a mapping type this is called as mapping

in elastic search. Mapping can be achieved implicitly relying on a server. Or can be defined explicitly by creating customized fields during index creation. Data stored inside the Elastic Search can be queried with Search API using REST requests or using Query DSL (Domain Specific Language) based on JSON to define the queries.

4.3 Kafka

Kafka[19] is divergent from a basic data storage concept, as it is used as a streaming platform. In Kafka a stream of messages of the particular type is defined by a topic. From Figure 2. Kafka architecture consists of a Producer – an application or a process which publishes messages to a topic, Broker – published messages are stored on a set of Kafka servers, Consumer – an application or a process which subscribes to one or multiple topics from brokers and pulls the data from them to consume.
Message in a Kafka is a basic form of communication between Producer and Consumer. Messages can be seen as a payload of bytes or any other format which is serialized from producer and consumer. In this experiment, we have selected Confluent Kafka for examining the performance which is a proprietary version of apache Kafka with some advanced features.

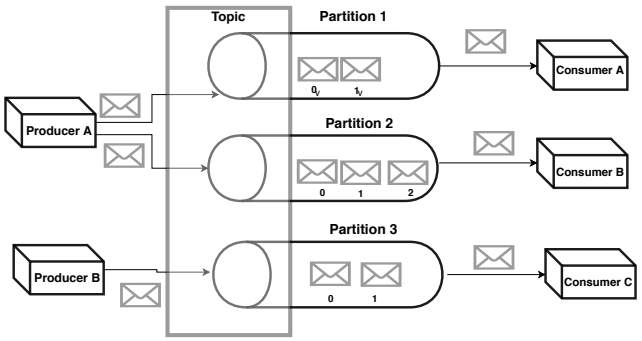

Figure 2: Kafka Architecture [15]

The overall architecture of Kafka is distributed in nature, a Kafka cluster consists of multiple brokers. To balance the load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.
In Figure 3. The records are assigned to partition in a sequential manner and each record has offset associated with it. Kafka record in partition persists until certain retention time period configured for that particular topic or until it is deleted. An offset is maintained by the consumer process, normally consumer advances its offset linearly but offsets can be reset and also can skip the records starting from presently inserted records. Giving offset control to the consumer makes Kafka more flexible to handle in terms of data retrieval.
[16]Querying the stored data from apache Kafka is not straightforward as it requires you to create a low-level query API using Kafka streams and table interfaces (KStream, KTable, and GlobalKTable) and this API is exposed to the outside world for performing interactive queries on the data, Kafka uses RocksDB storage engine

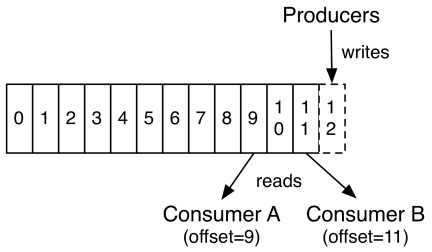

Figure 3: Kafka Read/Write to Partition [30]

internally. There are two possibilities of implementing the API first 'local state store' which limits querying the only local state of the records and second 'remote state stores' which allows sharing the state of the record by introducing the RPC endpoints. This is a limitation of the apache Kafka. To overcome this we choose to perform the tests on a proprietary variant of apache Kafka which is called [17]Confluent Kafka. We are using a free development version of Confluent Kafka for this experiment. Confluent Kafka is easy to maneuver in terms of querying the data using KSQL which is ready to use querying utility in Confluent Kafka.

4.4 MongoDB

MongoDB is famous document-oriented open-source NoSQL database.MongoDB data model consists of document as a smallest unit of data. Group of similar documents is called as a 'Collection' which can be seen as a table in traditional RDBMS. Single host of MongoDB can have multiple databases which can accommodate multiple collections.documents are stored by their semantics in collections and references can be used to link the documents to each other.
MongoDB supports Master-Slave replication and special replica set. Major difference between replica set cluster and master-slave cluster is in replica set there is no fixed master node, instead nodes in replica set conducts election to select the master node which is called 'Primary' and other nodes are called 'Secondary' which are analogous to slave nodes. It helps as a fail-over mechanism in MongoDB also secondaries can be used to reduce the overhead on master node to serve the requests.
Figure 4. is basic architecture of MongoDB, components are explained below: Sharding is used for scaling out in MongoDB. Using sharding collection data is split into chunks and distributed over the machines which helps to minimize the load and handle horizontal growth of data.MongoDB supports autosharding, there is 'range-based sharding' and 'hash based partitioning'.'Mongod' instance is responsible for the data storage and sharding on the other hand 'mongos' acts as a router to locate the data stored.config server is used to hold all the configuration of MongoDB cluster.
Write concern is used to receive the acknowledgement of the data inserts, update, deletes. When write concern value is low, level of guarantee that operation is successful is low and vice versa. Value for write concern varies from zero to number of nodes. There is also special 'majority' flag which returns when minimum number of data bearing nodes successfully performs the operation. Read Preference, is used to decide which node serves the requests, default is primary node, where all the read operations are carried out through

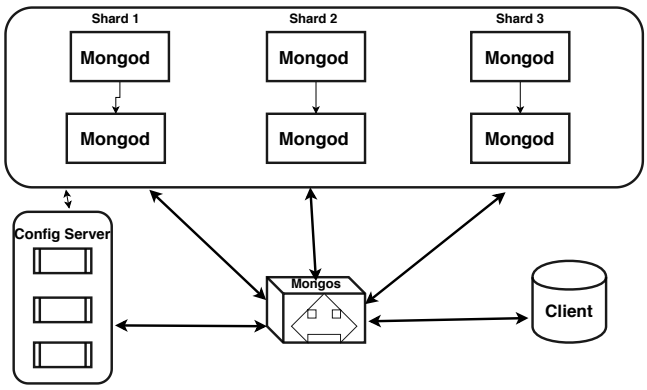

Figure 4: MongoDB Architecture

primary node. Possible values for read preference are – 'secondary', 'secondaryPreferred','nearest','primaryPreferred'. By configuring write concern and read preference as per data read/write criticality, we can achieve good performance.

5 EXPERIMENTAL SETUP

All four platforms mentioned in section 4 are installed on a separate Debian 9 virtual machines deployed with 2GB memory, 15GB disk memory, one CPU, two Network Adapters on virtual box. VMs are also equipped with python 3.5, jupyter notebook, pandas, seaborn and GitHub along with respective python library for each storage platform.

Initially, all the performance reading are taken just after the basic installation of all the storage platforms. Performance of read and write are the only scope of this experiment hence we have not performed any update or delete queries and recorded their individual performances.

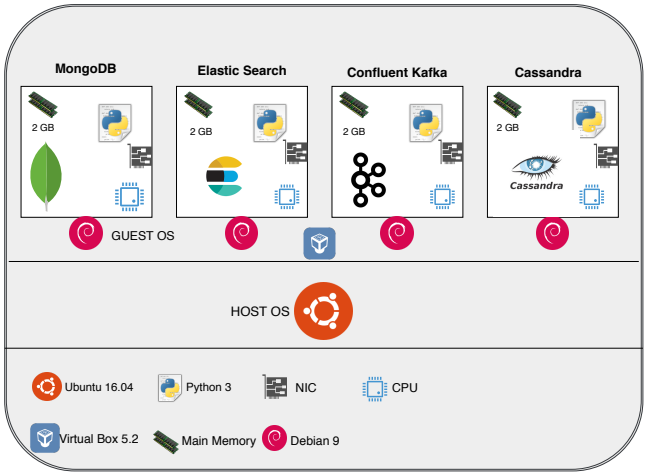

Figure 5: Basic Experimental Setup

After installation of an individual platform, we have inserted JSON record using python library of that particular database and recorded the performance of write operation. JSON records which are being inserted changes the values of the fields dynamically and also has an ID field which acts as a unique identifier of that JSON document.

Here is the structure of the JSON records we are inserting, we have assumed the data we are inserting is a stream of logs which is generated from the virtual machines after triggering some events.

```
document_json = {
                'id' : 5bd5c0ed6e955219e093bc33,
                'syscall_nr':30,
                'syscall_name':lalryu,
                'dtb': '0x0faf0689a9a64759',
                'rsp':'0xe675808c98ca4518',
                'rip':'0x68be8a3fcd604d73',
                'pid':3000,
                'vmid':'vm24',
                'logtype':'yxx',
                'ts':NumberLong("1545144718615")
                }
```

6 PERFORMANCE EVALUATION

There are two steps evaluation of each database, first gathering the performance readings based on default configurations and later optimizing configuration and improving the code to read and write the data.

6.1 Default Configuration

After installing all the databases on their respective VMs, we manually created 'Database' and 'Collection' in MongoDB, 'Topic' in confluent Kafka, 'Keyspace' and 'Column' in Cassandra and 'Index' in Elastic Search. We run the jupyter notebook which initially populates all the databases with a bunch of records for incremental order of 10 seconds till 1 minute and also retrieves the records based on its vmid in the same way. Finally, we take an average of the results to arrive at the single constant value for that database.

Below section describes all the plots taken for respective databases. All the plots below are the last iteration with an increment of 10 seconds.

6.1.1 MongoDB Basic - Figure 6. Average of all the iterations is 2,742 inserts/sec and 70,790 reads/sec

6.1.2 Cassandra Basic - Figure 7. Average of all the iterations is 314 inserts/sec and 3,811 reads/sec

6.1.3 Elastic Search Basic - Figure 8. Average of all the iterations is 106 inserts/sec and 60 reads/sec So far, these are lowest values.

6.1.4 Kafka Basic - Figure 9. Average of all the iterations is 54,032 inserts/sec and 692 reads/sec

6.1.5 Observations and Analysis. Graphs from section 6 are based on the basic database setup. comparing all readings it clearly shows that, Confluent Kafka from Figure 9. has highest – 54,032 inserts/sec as well as MongoDB from Figure 6. has highest – 70,790 reads/sec. On the other hand, the lowest read and write performance is given by Elastic Search from Figure 8. It can also be seen that Write performance of Confluent Kafka from Figure 9 and Elastic Search

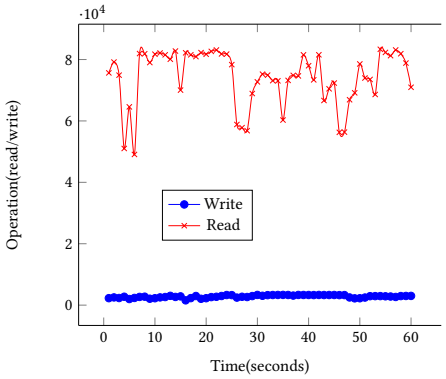

Figure 6: MongoDB

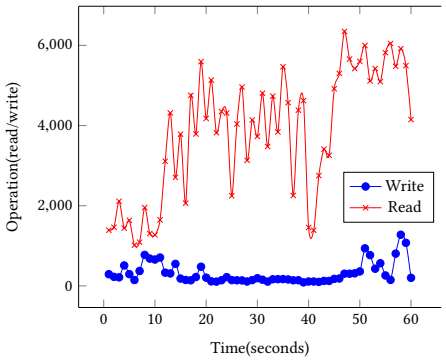

Figure 7: Cassandra

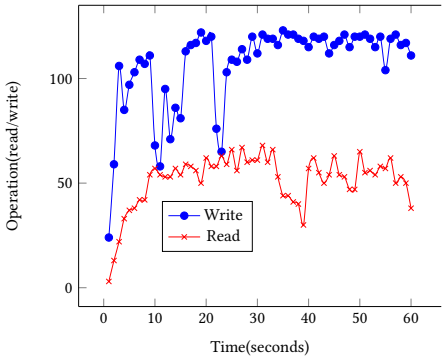

Figure 8: Elastic Search

from Figure 8 is more than read. In contrast, Read performance of MongoDB from figure 6 and Cassandra from figure 7 is more than write. Improvement in terms of code as well as the configuration is required to achieve good results in terms of read latency and write throughput.

6.2 Optimized Configuration

There are a variety of avenues to improve the performance of the database. There can be write intensive applications which require

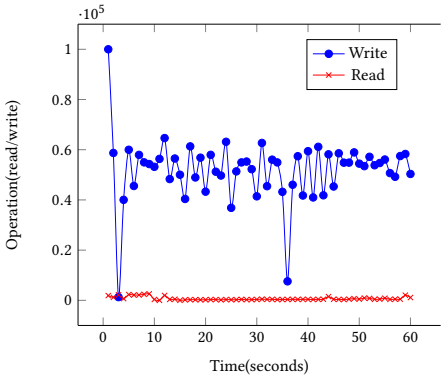

Figure 9: Confluent Kafka

the fastest writes as compared to reads. For example – writing real-time log data of multiple virtual machines in a cloud environment. On the other hand, there can be read heavy applications which executes multiple queries in parallel. For example – various reporting applications which performs read, mathematical and logical operations on stored data.

There are three ways for gaining good performance results, first – Improve and Optimize Code and Queries, second – optimize Database Configuration. The third way which can be upgrading hardware. For example, using SSD (Solid State Drive) [39] for journaling or storing the latest data often termed as 'Hot Data'. But, this can affect the cost of the project, therefore, improving quality by learning databases would be a wise and efficient decision.

Improvements in coding can be achieved by multiprocessing[38], multiprogramming[34], bulk or batch operations, optimize the query for faster data retrieval.

Also, Second way can be fine-tuning the configuration parameters. For example, in MongoDB, we have 'writeconcern' which can be set to lower value to gain good write performance. And 'readpreference' results in good read performance by redirecting the queries straight towards the node mentioned in the configuration without contacting the primary node. [22]

Below sections describe various optimizations done on databases along with their improved performances results.

6.2.1 MongoDB Optimized. From Section 6.1.1 it is seen that write performance of the MongoDB is too low as compared with the read performance. Hence, configurations which lead to good write performance are needed to apply.

There are various pythonic APIs available to interact with MongoDB. For example pymongo[27], pyMODM[26] and Mongoengine[25]. For this experiment, we are using pymongo, as it is the official driver published by Mongo developers and also provides good performance in this use-case compared to mongoengine.

To improve write performance we are using 'insert_many' method which supports bulk insert in pymongo. During database connection setup we can assign values to WriteConcern[24] instance using MongoClient method used to establish the database connection.[27] Below is the syntax of WriteConcern and how it is used in the experiment.

```
class pymongo.write_concern.WriteConcern(w=None,
    ↪ wtimeout=None, j=None, fsync=None)

from pymongo import MongoClient
client = MongoClient('localhost', 27017,w=0)
```

Parameters:
w: (integer or string) w=<integer> always includes the replica set primary (e.g. w=3 means write to the primary and wait until replicated to two secondaries).
wtimeout: (integer) Used in conjunction with w. Specify a value in milliseconds to control how long to wait for write propagation to complete. If replication does not complete in the given timeframe, a timeout exception is raised.
j: If True block until write operations have been committed to the journal. Cannot be used in combination with fsync.
fsync: If True and the server is running without journaling, blocks until the server has synced all data files to disk.[27]
As we are less concerned with the data integrity with one mongo node and want to have high performance of write operation we set a value of w to zero. It disables acknowledgment of write operations and can not be used with other write concern options (wtimeout,j and fsync). Also, we are using 'WiredTiger' Storage Engine which does not require journaling to guarantee a consistent state after a crash. It is default storage engine in MongoDB 3.2 and later versions. WiredTiger also provides a document-level concurrency model, checkpointing, and compression.[22]There is also 'In-memory' storage engine which ensures low latency by avoiding disk I/O but this is supported in MongoDB enterprise version.
There are two different data-structures used in WiredTiger to store the data, first 'B-tree' and second 'LSM'(Log Structured Merge) there are some benchmarking experiments which proves that LSM provides high write throughput than B-Tree and hence can be used in write-heavy applications. But unfortunately, LSM implementation of the WiredTiger is not supported in the recent MongoDB but Mongo developers are adding it in upcoming versions. [9][10]
Existing read performance of MongoDB can also be improved by creating indexes on the specific columns in the table. By default, mongo assigns '_id' it is treated as a primary key called Object ID, which is unique for each document in a collection. Also, an index is automatically created on this field. Other than default index there are many types of index configurations supported in MongoDB – Single Field, Compound, Multikey, Geospatial, Hashed Indexes.[22]
In Addition to write concern and index creation we have also applied snappy block compression which is provided by WiredTiger.
Monitoring database performance is very crucial as there are many ways in which the performance of the database might degrade. Therefore, MongoDB provides basic monitoring commands – 'mongotop' it shows how much time was spent reading or writing each collection over the last second. 'mongostat' gives lots of information on MongoDB instances. There are some visualization tools – 'MMS' (MongoDB Management Service) this is not just a monitoring tool but Backups, Replica Set Configuration, Sharded Cluster Deployments, and Upgrades.[23]
Below section shows performance evaluation after applying optimizations.

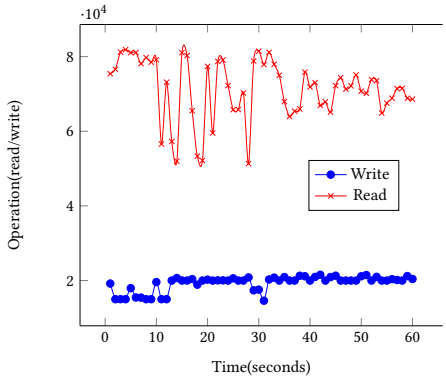

Figure 10: MongoDB

It can be clearly compared with the graph from Figure 6 that Write performance has gained some values and on the other hand read performance is also the same as earlier. Average write performance recorded was 22,020 inserts/sec and read 73,323 reads/sec. Therefore, it can be clearly seen that there is 10 times better write performance achieved after optimization.

6.2.2 Cassandra Optimized. Cassandra provides a stress tool called 'Cassandra-stress', this tool is an effective tool for populating a cluster and stress testing CQL tables and queries. There are lots of options by which we can perform the stress tests, for our experiment YAML-based profile stress test is suitable. A yaml-based profile is used for defining specific schemas with various compaction strategies, cache settings, and types. Also, a YAML file can be used to design tests of reads, writes, and mixed workloads. With the use of yaml profile, we can create different schemas, with different configurations and working environment. This tool also generates nice graphs after benchmarking the performance.[3]
As part of optimization, all the changes in configurations can be applied in the single yaml file, which is flexibly executing in different modes.
Initially, understanding how the YAML-profile is organized.
DDL – for defining your schema
Column Distributions – for defining the shape and size of each column globally and within each partition
Insert Distributions – for defining how the data is written during the stress test
DML – for defining how the data is queried during the stress test
After putting everything together we can pass the YAML file to Cassandra-stress with one of three different choices to of operations(Inserts, Queries and Mixed) along with other optional parameters.[20]
Below is the YAML Profile we are using for an experiment.

```
keyspace: perftesting
keyspace_definition: |
  CREATE KEYSPACE perftesting WITH replication = { '
      ↪ class':'SimpleStrategy','replication_factor
      ↪ ': 1};
```

```
  table: cassandra_test3

table_definition: |
    CREATE TABLE perftesting.cassandra_test3 (
    log_id text,
    vmid text,
    dtb text,
    logtype text,
    pid int,
    rip text,
    rsp text,
    syscall_name text,
    syscall_nr text,
    value int,
    PRIMARY KEY (log_id, vmid)
    )WITH CLUSTERING ORDER BY (vmid ASC)
    AND compression = {'chunk_length_in_kb': '4', '
        ↪ class': 'org.apache.cassandra.io.compress.
        ↪ LZ4Compressor'}
insert:
  partitions: fixed(1)
  batchtype: UNLOGGED
queries:
  read1:
    cql: select * from cassandra_test3 where vmid = ?
        ↪  ALLOW FILTERING
    fields: samerow

./cassandra-stress user profile=/root/stress.yaml n
    ↪ =150000 ops(insert=1) -rate threads=1 -graph
    ↪ file=test_write.html title=Perf_Write
    ↪ revision=Write1 -log file=write_$NOW.log
```

In the above profile, a table is created called cassandra_test3 with compression property set to LZ4Compressor with 4kb chunk size. Also, write consistency and read consistency which is by default set to one as there is only one Cassandra node. Unlike MongoDB there is no zero value which interprets no acknowledgment.
default compaction strategy – Size-tiered Compaction Strategy (STCS) is also best suited for the experiment because STCS is triggered when the system detects that there are enough (four by default) similarly sized SSTables. Once triggered, the tables are merged, resulting in one larger SSTable. As time progresses and several large SSTables have accumulated, they will be merged to form one even-larger SSTable and so on. As per, Scylla DB which is compatible with Cassandra claimed to be achieving high performance. STCS compaction is good for write performance improvement.[29]
Also clustered the 'vmid' which are limited to 16 different ids in ascending order, which will help to read the data more efficiently from the database.
Batchtype in Cassandra is used to ensure the atomicity and isolation of the DML operations. If we set the batchtype as LOGGED then writing the rows to the batchlog will be additional overhead and will penalize the performance. Therefore, batchtype defined in the insert section of the profile is set to UNLOGGED.[2]

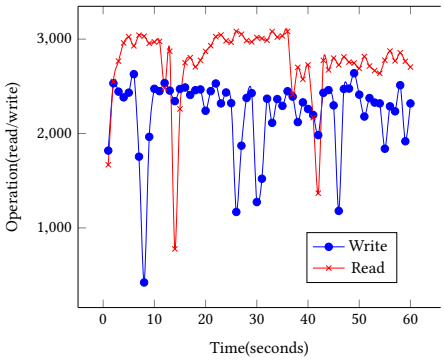

Figure 11: Cassandra

Performance test can be executed by the command at the bottom in code. In command, we are performing inserts with the number of rows added would be 1,50,000 and this operation is performed with only 1 thread.
Results in Figure 11 are retrieved from the stress tool along with the optimization parameters applied. From the graph Figure 11 and Figure 7 it can be inferred that there is an improvement in write performance from 314 inserts/sec to 2,203 inserts/sec. There is a slight drop in read performance is recorded where earlier it was 3,811 reads/sec and now it is 2,769 reads/sec.

6.2.3 Elastic Search Optimized. From the basic configuration, it can be seen that Elastic Search is the lowest performing database among all. As an imporovements implemented the bulk inserts in the index using the bulk method provided by the python driver.
The bulk API that provides a more human-friendly interface, it consumes an iterator of actions and sends them to elasticsearch in chunks. It returns a tuple with summary information like number of successfully executed actions and either list of errors or number of errors if stats_only is set to True.[4]
Below is the syntax of the bulk method.

```
elasticsearch.helpers.bulk(client, actions,
    ↪ stats_only=False, *args, **kwargs)
```

We have performed a number of tests for choosing the size of actions array and finally, well-performing size is 1,500 docs per action.which is generated by gendata method in a playbook.[6]

```
bulk(es, gendata(), stats_only=False)
```

We have also increased the refresh interval which is used to create segments and it will be available for the search within an index after a given time interval. default value of refresh interval is set to one second from elastic documentation of indexing performance tuning it is recommended to increase it to 30 seconds to achieve good indexing speed. Therefore, we have altered index.refresh_interval to 30s.[6] There are additional recommendations provided in the manual which does not cover the scope of experiments hence they are skipped.

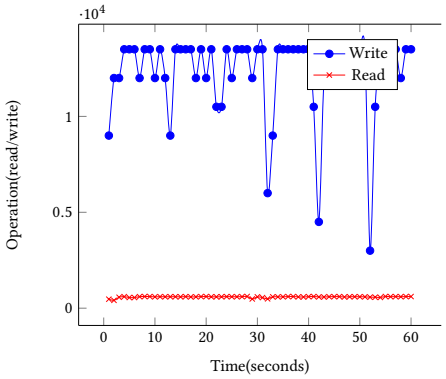

Figure 12: Elastic Search

Search operation takes place in two phases first is Query and the second one is Fetch. In query phase after accepting the query, node identifies the index in that are being searched and finds out the shards which contain data for that index. A query is then processed and scored by the shards further in fetch phase all the scored top document's results are fetched. One way to improve the performance of your searches is with filters. With a filtered query, working with boolean matches, you can search for all documents that contain X before scoring on whether or not they contain Y. Also, filters can be cached. Therefore, we have applied a filter to our query which will only give the vmids for logtype of value 'xxy' which will automatically filter the other logytypes and improve the query performance.
Figure 12 is a performance graph for elastic search.
From the evaluation of Elastic Search in Figure 8 both read and write operation has the lowest performance. After optimization write performance has increased exponentially from 106 writes/sec to 12,300 writes/sec along with read from 60 reads/sec to 584 reads/sec in Figure 12.

6.2.4 Kafka Optimized. Section 4.3 overviews the basics about Kafka platform. For an experiment, selection of Confluent Kafka as a storage solution is due to a few promising features confluent Kafka provides. The first major advantage of python client which apache Kafka does not support. It also provides KSQL which is a streaming SQL engine which provides powerful SQL like interface to process the stream data. It can be used for transformation and parsing of the stream data, joins, and aggregations. There are lots of additional features offered by open source version of confluent Kafka along with enterprise version.
From the initial basic Kafka setup, we could achieve great throughput for insert operation. The most important step we can take to optimize throughput is to tune the producer batching to increase the batch size and the time spent waiting for the batch to fill up with messages. Larger batch sizes result in fewer requests to the brokers, which reduces a load on producers as well as the broker CPU overhead to process each request. For Python client we have set 'batch.num.messages' to 2,00,000. This property and the bulk insert are different optimizations applied.
Also, enabled compression by configuring the 'compression.type' parameter. Compression means a lot of bits can be sent as fewer compressed bits, according to the compression algorithm used. Kafka supports lz4, snappy, and gzip. Compression is applied on full batches of data, so the efficacy of batching will also impact the compression ratio. The number of acknowledgments the producer requires the leader to have received before considering a request complete is set to zero. Which means there is no acknowledgment propagated to the leader.[1]
produce() is asynchronous, all it does is enqueue the message on an internal queue which is later served by internal threads and sent to the broker. Adding flush() before exiting will make the client wait for any outstanding messages to be delivered to the broker. If we add flush() after each produce() call we are making it a synchronized producer which is performance killer. Therefore, polling just after produce call is efficient and later we can handle the message retries by again polling and flushing
consumers higher throughput is adjusted on basis of volume of data it gets from each fetch from the broker. By increasing the configuration parameter 'fetch.min.bytes' to 1,00,000 reduce the number of fetch requests made to the broker, reducing the broker CPU overhead to process each fetch, thereby also improving throughput. Broker and Consumer default configuration values are already optimized hence, we have kept it unchanged. To test the different batch size of bulk inserts batch sizes varying from 1000 to 60000 are applied and results are projected further to rate per second. After getting the results in Figure 13 the largest throughput batch size which is 18000 for a write operation and 49000 for reading. Further performed a regular 60 seconds performance test on that particular batch size in Figure 14.

```
for i in range(msg_count):
try:
producer.produce(topic, value=msg)
producer.poll(0)
except BufferError as e:
producer.flush()
to_retry += 1

#polling messages that overflowed the local buffer
for i in range(to_retry):
producer.poll(0)
try:
producer.produce(topic, value=msg)
except BufferError as e:
producer.poll(0)
producer.produce(topic, value=msg)

producer.flush()
```

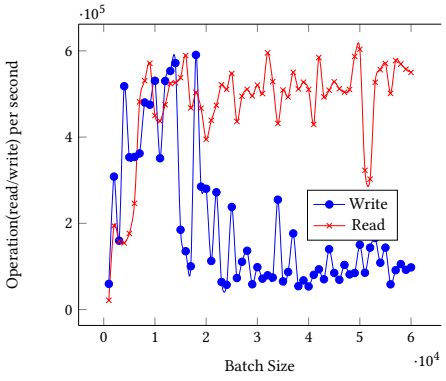

Figure 13: Kafka Variable Batch Size

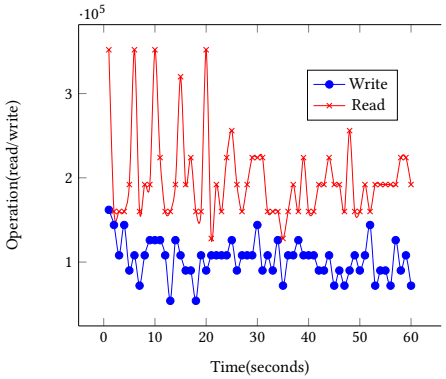

Figure 14: Kafka

7 CONCLUSION

In this paper, some common improvements which have been done on the coding part are bulk inserts implementation which has helped to boost the write performance significantly. For read performance query optimization has been helpful. For example, specifically for elastic search with a filtered query, working with boolean matches, you can search for all documents that contain filter condition before scoring on whether or not they contain other values instead of the condition. Also, filters can be cached.[28]
Another factor is using compression during database writes. But, there are lots of factors which affects the compression algorithms. for example compression and decompression time, compression ratio and CPU cycles consumed to perform the compression. based on all the above factors best compression algorithm is applied which eventually turned beneficial for good performance.[41]. The concept of write acknowledgment exists in all the databases and it is configurable in different ways as per the database architectures and data models. This alteration is helping to achieve good write performance. Looking at the nature of data in the present day we can certainly agree that choosing the database solution and optimizing it is the basic need. As a result of the experiment carried out in this paper Kafka provides a significant read and write performance compared with others. But, there are some fallbacks or limited features in Kafka which makes it just a streaming platform instead of a database management platform. By seeing at the data model and the features Kafka has to offer, it is more suitable for the time series data which can be transformed and processed in near real-time manner. querying and storing historical data in Kafka can get tricky sometimes.

8 FUTURE SCOPE

Experiment performed in this paper is divided into the basic configuration comes with the database installation and further each database is optimised for read and write performance. There is wide scope of optimizations which can be applied and this paper focuses on some of them. Also, setup of this experiment is not same as the huge production setup which can have several performance optimisation avenues.
As a result of this expermient it is seen that kafka is good with read and write performance. But, kafka is not a database management platform. Therefore, to transform and process the humoungous data into useful in real-time kafka can be integrated with normal databases to serve the queries. Finally, design of a system based on above assumptions boils down to kappa-architecture which is treated as a simplification of the lambda architecture without batch processing layer. This can be taken as a second step into the experiment which would test the different platform's performances in kappa architecture.

REFERENCES

[1] Yeva Byzek. May 10, 2017. Optimizing Your Apache Kafka Deployment. https://www.confluent.io/blog/optimizing-apache-kafka-deployment/.
[2] Cassandra. 2018. Batch. https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlBatch.html.
[3] DataStax. 2019. The cassandra-stress tool. https://docs.datastax.com/.
[4] ElasticSearch. 2018. Bulk helpers. https://elasticsearchpy.readthedocs.io/en/master/helpers.html.
[5] ElasticSearch. 2018. Elasticsearch Reference 6.5. https://www.elastic.co/guide/en/elasticsearch/.
[6] ElasticSearch. 2018. Tune for indexing speed. https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html.
[7] Andrea Gandini, Marco Gribaudo, William J Knottenbelt, Rasha Osman, and Pietro Piazzolla. 2014. Performance evaluation of NoSQL databases. In *European Workshop on Performance Engineering*. Springer, 16–29.
[8] GitBook. 2018. Understand the Cassandra data model. https://pandaforme.gitbooks.io/introduction-tocassandra/.
[9] Alex Gorrod. 2017. Btree vs LSM. https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM.
[10] Alexander Gorrod. 2018. Expose LSM configuration options to the engine, index and collection creation commands. https://jira.mongodb.org/browse/SERVER-18396.
[11] Jing Han, Haihong E, Guan Le, and JianDu. 2011. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*. 363–366.
[12] R. Hecht and S. Jablonski. 2011. NoSQL evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*. 336–341. https://doi.org/10.1109/CSC.2011.6138544
[13] Eben Hewitt. 2011. *Cassandra the definitive guide*. O'Reilly.
[14] M. Jung, S. Youn, J. Bae, and Y. Choi. 2015. A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment. In *2015 8th International Conference on Database Theory and Application (DTA)*. 14–17. https://doi.org/10.1109/DTA.2015.14
[15] Apache Kafka. 2018. Apache Kafka distributed streaming platform. https://kafka.apache.org/intro.
[16] Apache Kafka. 2018. Developer Manual. https://kafka.apache.org.
[17] Confluent Kafka. 2018. KSQL. https://docs.confluent.io/current/ksql.
[18] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. 2015. Performance Evaluation of NoSQL Databases: A Case Study. In

Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems (PABS '15). ACM, New York, NY, USA, 5–10. https://doi.org/10.1145/2694730.2694731

[19] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 1–7.

[20] Jake Luciani. 2014 July. Improved Cassandra 2.1 Stress Tool: Benchmark Any Schema. https://www.datastax.com/dev/blog/improved-cassandra-2-1-stress-tool-benchmark-any-schema.

[21] Bernard Marr. 2018. Here's why Data is not the new Oil. https://www.forbes.com/sites/bernardmarr/2018/03/05/heres-why-data-is-not-the-new-oil.

[22] MongoDB. 2018. The MongoDB 4.0 Manual. https://docs.mongodb.com/manual/.

[23] MongoDB. 2018. MongoDB Monitoring Service Manual. http://api.mongodb.com.

[24] MongoDB. 2018. Write Concern. https://docs.mongodb.com/manual/reference/write-concern/.

[25] Mongoengine. 2018. Mongoengine. http://mongoengine.org/.

[26] PyMODM. 2018. Getting Started with PyMODM. https://pymodm.readthedocs.io/en/latest/getting-started.html.

[27] Pymongo. 2018. PyMongo Documentation. https://api.mongodb.com/python/.

[28] Paul Rossmeier. 2018. Elasticsearch Query. https://www.objectrocket.com.

[29] Scylla. 2018. Compaction Strategies. https://docs.scylladb.com/architecture/compaction/compaction-strategies/.

[30] Abhishek Sharma. 2014. Apache Kafka: Next Generation Distributed Messaging System. https://www.infoq.com/articles/apache-kafka.

[31] Wikipedia contributors. 2018. Apache Cassandra — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=871249223. [Online; accessed 20-December-2018].

[32] Wikipedia contributors. 2018. Apache Kafka — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=874134311. [Online; accessed 20-December-2018].

[33] Wikipedia contributors. 2018. Apache Lucene — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Lucene&oldid=871301554. [Online; accessed 20-December-2018].

[34] Wikipedia contributors. 2018. Computer multitasking — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Computer_multitasking&oldid=867491301. [Online; accessed 20-December-2018].

[35] Wikipedia contributors. 2018. Database model — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Database_model&oldid=870620584. [Online; accessed 20-December-2018].

[36] Wikipedia contributors. 2018. Elasticsearch — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Elasticsearch&oldid=873349378. [Online; accessed 20-December-2018].

[37] Wikipedia contributors. 2018. MongoDB — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=MongoDB&oldid=874538490. [Online; accessed 20-December-2018].

[38] Wikipedia contributors. 2018. Multiprocessing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Multiprocessing&oldid=870516982. [Online; accessed 20-December-2018].

[39] Wikipedia contributors. 2018. Solid-state drive — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Solid-state_drive&oldid=873952228. [Online; accessed 20-December-2018].

[40] Elizabeth Woyke. 2018. Breakthrough Technologies Sensing City. https://www.technologyreview.com/lists/technologies/2018/.

[41] Peter Zaitsev and Vadim Tkachenko. 2016. Evaluating Database Compression Methods. https://dzone.com/articles/evaluating-database-compression-methods.