Spring 5-24-2021

NewSQL Monitoring System

Akash Budholia

NewSQL Monitoring System

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfilment

Of the Requirements for the Degree

Master of Science

By

Akash Budholia

May 2021

The Designated Project Committee Approves the Project Title

NewSQL Monitoring System

by

Akash Budholia

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

MAY 2021

Dr. Suneuy Kim	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Mr. Omkar Malusare	Software Engineer 3 at Walmart Labs

ABSTRACT

NewSQL Monitoring System

By

Akash Budholia

NewSQL is the new breed of databases that combines the best of RDBMS and NoSQL databases. They provide full ACID compliance like RDBMS and are highly scalable and fault-tolerant similar to NoSQL databases. Thus, NewSQL databases are ideal candidates for supporting big data and applications, particularly financial transaction and fraud detection systems, requiring ACID guarantees. Since NewSQL databases can scale to thousands of nodes, it becomes tedious to monitor the entire cluster and each node. Hence, we are building a NewSQL monitoring system using open-source tools. We will consider VoltDB, a popular open-source NewSQL database, as the database to be monitored. Although a monitoring dashboard exists for VoltDB, it only provides the bird's eye view of the cluster and the nodes and focuses on CPU usages and security aspects. Therefore, several components of a monitoring system have to be considered and have to be open source to be readily available and congruent with the scalability and fault tolerance of VoltDB. Databases like Cassandra (NoSQL), YugabyteDB (NewSQL), and InfluxDB (Time Series) will be used based on their read/write performances and scalability, fault tolerance to store the monitoring data. We will also consider the role of Amazon Kinesis, a popular queueing, messaging, and streaming engine, since it provides fault-tolerant streaming and batching data pipelines between application and system. This project is implemented using Python and Java.

ACKNOWLEDGMENT

I would like to express my gratitude to my advisor Dr. Suneuy Kim for her constant guidance and support throughout the project. Her advice and motivation have been instrumental in the completion of this project. I also thank the members of my committee Dr. Robert Chun and Mr. Omkar Malusare, for their timely input and feedback, which were extremely helpful in enhancing the project. Finally, I would like to thank my family and friends for their never-ending support, motivation, and encouragement.

Table of Contents

LIST OF TABLES:

LIST OF FIGURES:

CHAPTER 1

Introduction

Data has become a vital and essential commodity in the past few decades, so much so that most organizations rely on data to gather insights and improve upon their customer experience and gain significant market share. Huge volumes of data are generated every day. Companies can use such data to identify trends that help them execute their strategies and make better predictions for demands and supply. Due to the advent of IoT applications and smart appliances like home automation devices, the volume of data generated has increased manifolds. Traditional means of storage and retrieval of data have proven inefficient in dealing with this influx of varied data of high volume and variety.

RDBMS has been the de-facto standard of data storage and retrieval for four decades. These worked well for applications with single node client-server architecture. However, with the advent of big data and microservice architecture, RDBMS could not meet the expectations of a low latency response to many concurrent requests [3]. As a result, there felt a need to look beyond the RDBMS to cater to the Big Data requirements. NoSQL databases could fulfill all these characteristics by providing features like scalability, replication for fault tolerance, and a flexible schema design to accommodate several data models like key-value pair (DynamoDB), document database (MongoDB), columnar databases (Cassandra), and graph databases (Neo4j).[3]

However, NoSQL databases prefer availability over consistency of data across a cluster of nodes in the event of failure, which becomes unacceptable when dealing with applications like financial transactions, fraud, and anomaly detection where strong consistency is a necessity. NoSQL databases sacrifice ACID guarantees (Atomicity, Consistency, Isolation, and Durability) in favor

of network partition and high availability and offer BASE (**B**asically **A**vailable, **S**oft State and **E**ventually consistent) properties [3]. Since ACID guarantees are critical for applications like e-commerce, companies have to add a sharding layer to their RDBMS systems to get the best of both worlds. However, sharded SQL databases are difficult to manage and require much expertise and manual effort to deploy. Therefore, there was a need to have databases built from the ground-up, which would combine the scalability and fault tolerance of NoSQL and provide ACID guarantees of RDBMS. NewSQL databases comply with both the above requirements. Most NewSQL databases reuse the existing open-source PostgreSQL or the MySQL query layer on top of persistent storage like RocksDB (a strongly consistent key-value store). NewSQL databases are promising, and the demand for this database is on the rise. Hence, it makes for a fascinating case to monitor such databases and evaluate which underlying database would be most efficient in supporting the monitored data of the NewSQL databases.

Since the research done in developing the monitoring system for NewSQL is a niche, we compare various database systems for monitored data management of the NewSQL database in this project. We have explored several candidate databases for each layer, weigh in their pros and cons, and choose the most appropriate databases to store the monitoring data. Organizations develop monitoring systems to keep their systems in an optimal state, keep track of resource utilization, detect anomalies in the system, and mitigate those issues by eliminating the process of manually monitoring every component of the database clusters. The features of a sound monitoring system are to provide an accurate representation of the current state of the system and produce low latency results as the monitoring data is generated at a rapid velocity. However, these monitoring systems are developed to cater to organizational needs and are not be open-sourced. Therefore, we aim to develop a NewSQL monitoring system from completely open-source technology. The

project's focus is to ensure that the components used to build the monitoring system offer the best performance and function efficiently with minimal manual intervention.

The project report is organized as follows: Chapter 2 provides an overview of the concepts related to the project. Chapter 3 focuses on the literature reviews, techniques, and previous works. The design and implementation of the monitoring system and the rationale behind the design decisions are explained in chapter 4. A few experiments were performed to study the efficacy of databases that store the data from the VoltDB cluster; chapter 5 focuses on the experimental setup, the results, and analysis. Finally, chapter 6 focuses on the conclusion and future scope possible with the project.

CHAPTER 2

Background

2.1 NoSQL Database

NoSQL emerged as a reaction to the non-flexibility and inability of the RDBMS database to scale in congruence efficiently with the increase in the size and the volume of data. The NoSQL databases were built to scale and support unstructured data. Most of the NoSQL solutions focus on availability by relaxing consistency requirements: the database is always available, but the queries to the different nodes in the cluster might yield different results. This form of data inconsistency is called eventual consistency. Several other NoSQL databases do not have a rigid schema. Key values stores are extended by replacing values with 'JSON' Documents which have sub-keys, sub-values, arrays of JSON objects, and hierarchies. [27]. Finally, graph databases organize data according to relationships instead of columns or rows to provide graph query features. The key feature of the NoSQL database is its ability to scale on inexpensive commodity hardware. Replication is the process of copying partition data across multiple nodes in the cluster to ensure high availability and fault tolerance. NoSQL databases usually follow two modes of replication:

- **Master-Slave Replication**: The writes are only directed to the master node and then replicated to the slave nodes, while any of the nodes can handle the read requests due to strong consistency. [12]
- **Peer-to-peer architecture**: Reads and write requests can be directed to any of the nodes in the cluster, and all the nodes have the same role. Data inconsistency may occur as these databases are 'eventually' consistent. Eventually, all replicas will have the latest write. [12]

CP (Consistent and Partition Tolerant) systems ensure strong data consistency, which means data is consistent across all the nodes, but the system may not be available entirely if some nodes in the cluster go down. NewSQL databases fall under the CP category.

AP (Available and Partition Tolerant) systems provide high availability but cannot guarantee data consistency across all the nodes. The data is written to one node, and there is no wait for the other node to come in agreement. Cassandra and DynamoDB fall under this category.

2.2 NewSQL Database

The term NewSQL databases was first coined by '451 Group' Analyst Matt Aslett [26] to emphasize a group of databases that share RDBMS functionalities and offer some of the functionalities of NoSQL databases. NewSQL provides the best of both worlds: the relational data model and ACID transactional consistency of traditional databases and scalability and speed of NoSQL databases. While RDBMS and NoSQL databases offer scalability options- they do it at the expense of missing out on transactional ACID guarantees and SQL standard interactivity. NewSQL databases like NoSQL provide high availability and fault tolerance, are cloud-compatible, and meet web-based application demands. While NoSQL databases prefer availability, a NewSQL database will always choose consistency in case of failure. [27] The NewSQL system will return the same answer to all the clients. Unlike NoSQL databases, they aim to maintain characteristics of relational databases. For example, the NewSQL query language is similar to SQL, and they provide ACID transactions.[5] They provide high performance, scalability, and

distribute-ability, which is achieved by leveraging improved algorithms and parallel computing, which were unavailable when RDBMS was designed three decades back. [7]

2.3 VoltDB

VoltDB is a fully-ACID compliant RDBMS that optimizes the use of modern computing environment. VoltDB performs in-memory computation, which avoids disk usage bottlenecks and maximizes the database throughput. It also provides serializable isolation, the strongest form of isolation, by eliminating costly and time-consuming processes like locking and latching. Each VoltDB database optimized the execution by partitioning the data and the corresponding stored procedures across multiple partitions across the cluster. The architecture of VoltDB makes it practical to process large streams of data efficiently and quickly and is mainly used in financial and IoT applications [8]. However, VoltDB is not optimal for business intelligence and uses cases where processing a large amount of historical data is required from multiple tables.

2.3.1 Partitioning in VoltDB

Each stored procedure is called a transaction and is atomic; either it succeeds or rollbacks completely. Since these stored procedures are written in Java, VoltDB can precompile the data access logic to distribute the data and its processing to individual partitions on the cluster [8]. Each node can handle multiple partitions. In single partitioned procedure execution, the server will execute the procedure by itself.

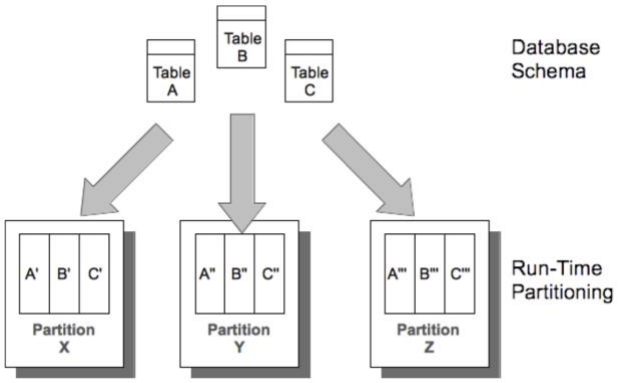

Figure 1. Load Partitioning in VoltDB [8]

If a stored procedure executes across multiple partitions, one node will act as a coordinator node, hands-off tasks to appropriate partitions, collect the results, and completes the tasks. The architecture of multiple partitions parallelly executing requests helps achieve maximum throughput. Each transaction runs in its thread and eliminates locking and latching, thus minimizing individual latency [4]. In addition, specific small tables can be replicated on all the cluster nodes, which helps perform joins between these tables while remaining single partitioned transactions.

2.3.2 Replication

'K-safety' is a mechanism that involves duplicating database partitions across multiple nodes based on the value of K, so that database can be made fault-tolerant in case one or more nodes fail in the cluster. These replica partitions are fully functional members and can handle read and write requests similar to a peer-to-peer model.

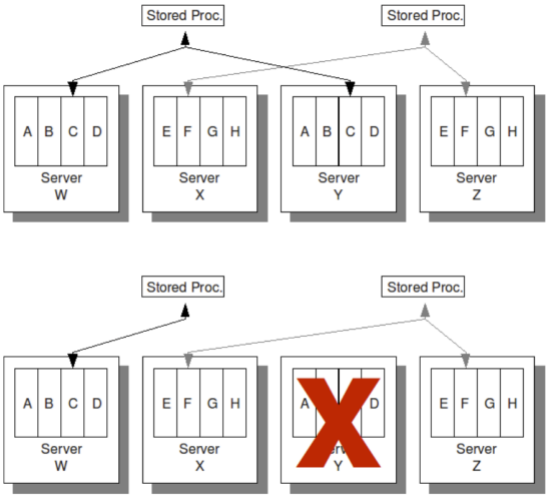

Figure 2. K-Safety in VoltDB [8]

If K = 1, it is a compulsion to duplicate all the partitions, and for K=2, it requires two duplicates of every partition in every node.

There is a direct correlation between the K-value and the number of nodes in the cluster and given by the following formula:

```
Unique partitions = (nodes * partitions/node) / (K + 1) [8]
```

2.3.3 Scalability

The design of VoltDB allows it to scale efficiently as per the needs of the applications that deploy it. Scaling of the VoltDB increases the throughput by increasing the number of request processing queues and the capacity (increasing the number of partitions). Scaling a VoltDB cluster does not require any changes to the database schema or the application code. A corresponding number of nodes need to be added to make a K-safe unit. E.g., For a K-safety value of 2, a total of three nodes must be added. [8]

2.4 Cassandra

Cassandra is a peer-to-peer, distributed NoSQL database designed to handle and store a massive amount of data on a multitude of commodity servers. Cassandra is inspired by Amazon's Dynamo replication technique and Google's BigTable data engine model [1][19]. Cassandra also offers its CQL (Cassandra Query Language), which has its roots in SQL. Cassandra follows a master-less architecture where initially one node acts as a coordinator node, but once the cluster is complete, every node can accept read and write requests.

Rows are organized inside tables where primary key(s) are the first columns followed by other columns in the lexicographical order [6]. The clustering key(optional) orders the partitions based on the order described by the user during table creation, and similar to RDBMS tables, may be dropped or altered at runtime with non-blocking read and writes. Cassandra does not support joins or subqueries and instead encourages denormalization through features like collections (maps, set, list).

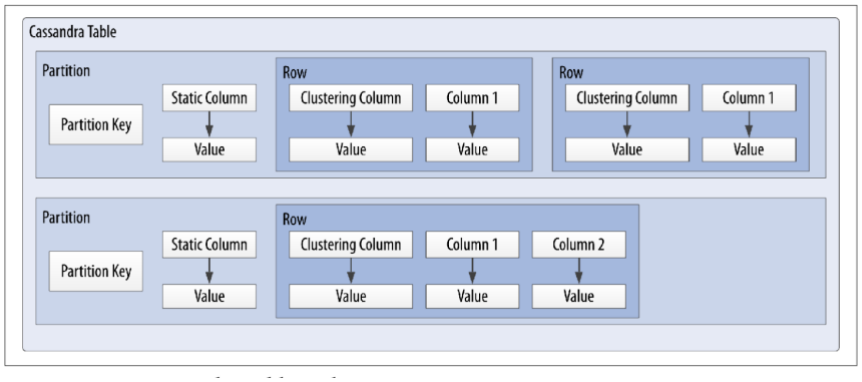

Figure 3. Cassandra Data Model [19]

A column family is similar to an RDBMS table, but unlike RDBMS, all the rows within the same row need not share the same column, making the Cassandra data model flexible and scalable.

Writes and read can be directed to any node and the node then recalibrates the request to the appropriate node, which has the corresponding data partition by applying a hash function [6]. Cassandra has a tunable consistency model where a 'consistency level' defines the number of replicas that need to be written before acknowledging the client application. Default consistency level is ONE, but more stringent consistency levels like QUORUM or ALL may be deployed to ensure appropriate data accuracy across a certain number of nodes [19]. Each Keyspace in Cassandra can have its replication strategy; there are two strategies to partition data.

- **Network Topology Strategy:** Allows replication factor to be declared for each data center explicitly, and replicas can be chosen to form different racks on the same data center [19]. If the number of racks is greater than the number of replicas, each replica will be chosen from a different rack to apply rack-aware behavior.
- **SimpleStrategy:** This strategy allows an integer replication factor to be defined at the time of keyspace creation and defines how many nodes should have a copy of the data. This replication scheme treats all the nodes as identical without any data center and rack configurations.

Cassandra deploys a consistent hashing scheme where data is randomly and evenly distributed across all the nodes of a cluster and retains the ease of adding and removing nodes from the cluster with minimal recalibration of data. Cassandra stores data in 'Memtable' and appends it to a commit log in case of a write operation. Memtable and SSTables, which are immutable, are for each table and append-only, but the commit log is shared across tables. A partition is stored across multiple SSTable files. [6]

2.5 YugabyteDB

YugabyteDB is a distributed, relational NewSQL database capable of handling a massive amount of organized data spanned across multiple availability zones. It provides high availability, low latency, and no single point of failure. YugabyteDB is a CP (Consistent and Partition Tolerant) database and offers very high availability in the CAP theorem spectrum. It also provides multi-row ACID transactions and offers Serializable and Snapshot Isolations. YugabyteDB reuses the SQL language in PostgreSQL under its YSQL and offers relational data modeling features like distributed transactions and referential integrity. Its YCQL API supports Cassandra compatible applications, with support for distributed transactions and strongly consistent secondary indexes. [20]

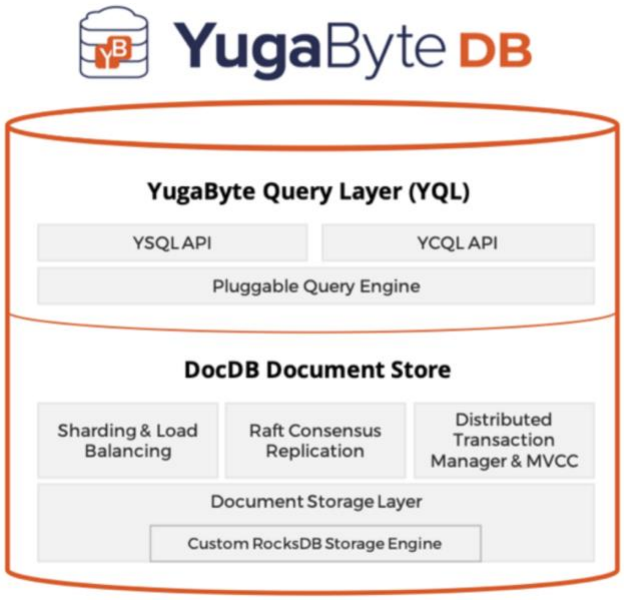

Figure 4. The architecture of YugabyteDB [21]

2.5.1 Architecture Components of YugabyteDB

2.5.1.1 YugabyteDB Query Layer:

Applications directly interact with the query layer using specific client drivers. Query Layer specifically deals with API-related aspects like query compilation and the runtime operations like data type representations, executing built-in operations.

The query layer consists of two essential aspects:

- **YB-TServer**

  YB-TServer handles end-user requests in the Yugabyte cluster. Tables are split into tablets, and each tablet has tablet-peers based on the replication factor.

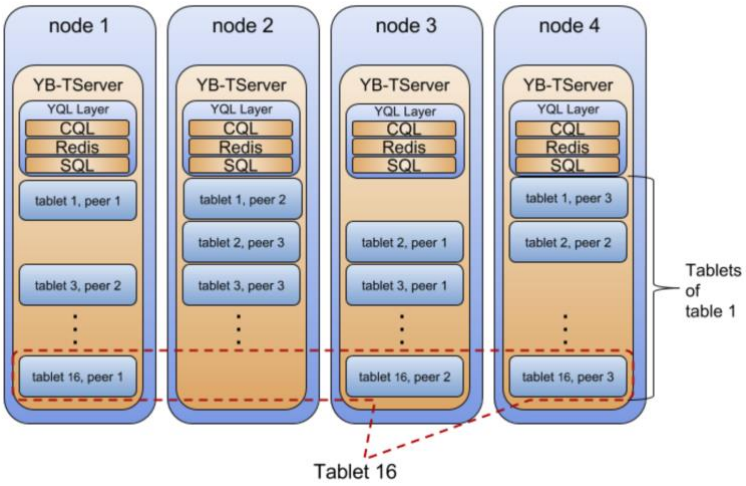

Figure 5. YB Tablet Server Service Architecture [21]

- **YB-Master**

  YB-Master maintains the system meta-data and records the tables in the system, user roles, permissions, each tablet's location. It performs load balancing and re-replication of under-replicated data and is highly available due to the formation of a Raft group. It is also responsible for administrative operations like creating, dropping, and altering tables.

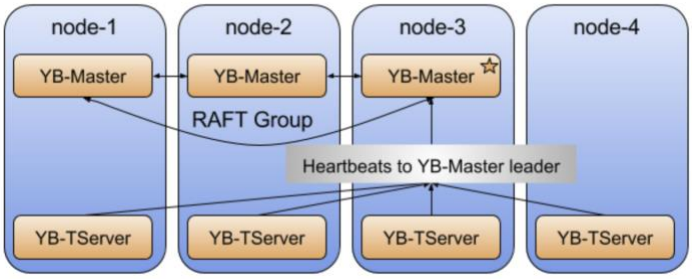

Figure 6. YB-Master Service Replication [21]

2.5.1.2 DocDB

DocDB is a distributed document stored inspired by RocksDB in its architecture. It provides automatic sharding, load balancing, strong write consistency, extreme resilience to failure, zone and rack awareness, and tunable read consistency. DocDB key consists of a key made of one or more hashed key components followed by range components and followed by MVCC timestamp in reverse order. Values in DocDB can be primitive (int32, float) or non-primitive (sorted maps) type.

```
DocumentKey1 = {
        SubKey1 = {
                SubKey2 = Value1
                SubKey3 = Value2
        },
        SubKey4 = Value3
}
```

Figure 7. DocDB Data Structure [22]

Every row is a document in DocDB, and the document key is the primary key with column values organized as a 16-bit hash of the column values followed by partition(hash) columns

followed by clustering(range) columns. The non-primary key columns are subdocuments, and the sub-document key is the columnID.

2.5.2 Sharding in YugabyteDB

YugabyteDB supports auto-sharding, is a highly available database, and supports hash and range sharding. Each shard is called a tablet, and it is placed on a corresponding tablet server.

- **Hash Sharding**

  In YugabyteDB, tables are allocated a hash space in a 2-byte range, accommodating up to 64K tablets in a huge dataset or cluster size. In read/write operations, the primary keys are converted into internal keys, and the hash value is calculated. E.g., if we want to insert a key k, a hash value of the key will be calculated, and the corresponding table will be looked up in the corresponding tablet server, and the request will be sent accordingly. [21]

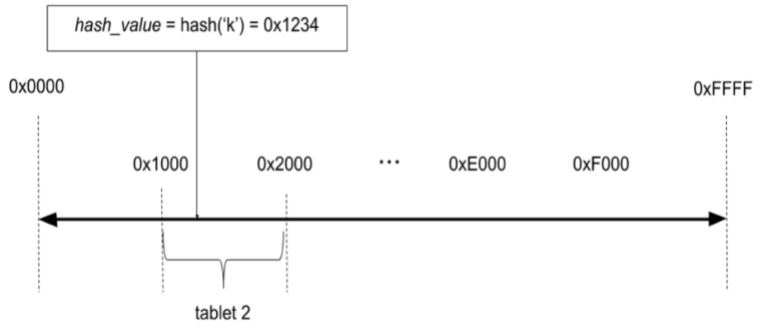

Figure 8. Hash Sharding in YugabyteDB [22]

- **Range Sharding**

Tables with ASC and DESC order defined for the first columns of a primary key and the first of the indexed columns cause the data to be stored in the chosen order in the single tablet. This tablet either automatically splits once it reaches a specific size or can be split manually. A good shard key for range sharding should have high cardinality, low recurring frequency. If the shards become too big, then optimal performance can be obtained by splitting the shard into multiple shards and rebalancing it across nodes. [21]

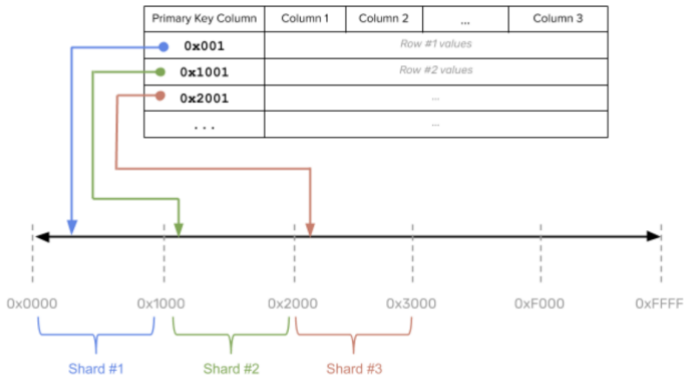

Figure 9. Range Sharding in YugabyteDB [22]

2.5.3 Replication in YugabyteDB

YugabyteDB performs synchronous data replication for fault tolerance and maintains data consistency using Raft consensus protocol. Replication is achieved via DocDB and every tablet-peer (which stores the replica of tablet data). The tablet peers are the same as the replication factor, and data between the tablet peers is strongly consistent. When a tablet is started, it elects one tablet leader using Raft protocol. It issues the user-issued commands into document storage of DocDB and replicates among the tablet-peers to maintain strong consistency.

2.6 InfluxDB

InfluxDB is an open-source time-series database developed in Go programming language by InfluxData and is optimized to handle time-series data. [14]

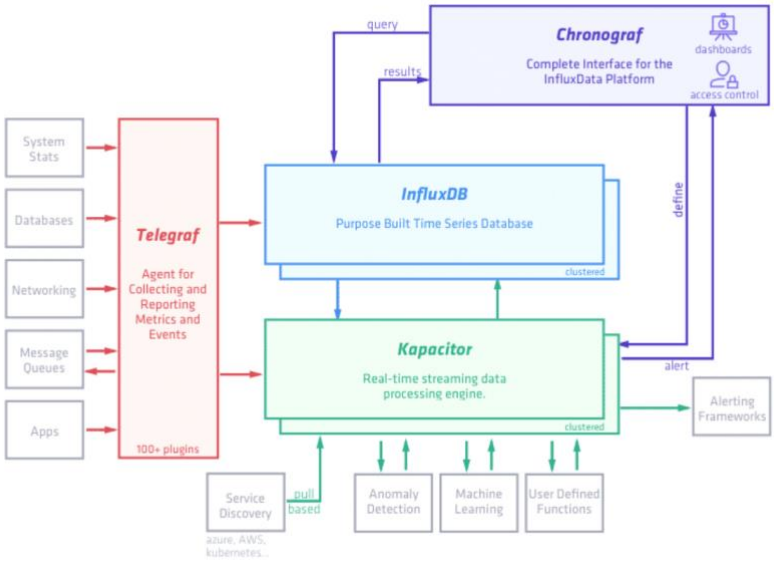

Figure 10. InfluxDB cluster Architecture [14]

Timestamp is included in every column of the InfluxDB database, where each timestamp is associated with a data point. The queries on tags are faster as compared to simple fields since tags are indexed. Measurement is another important concept which is equivalent to SQL table and explains field content. The retention period defines how long InfluxDB retains the data, with default retention being infinite and replication factor 1.

2.6.1 Sharding in InfluxDB

Sharding is the horizontal partitioning of data in InfluxDB. Data is stored in shard groups organized by retention policies and data and timestamps within that time intervals. The default duration is one hour for two days, one day for the duration between two days and six months, and

seven days for durations beyond six months. Optimal shard duration selection is essential for efficient drop operations since data is dropped on every shard basis. Too short shard duration can cause data compression issues. For efficient compaction, the storage engine groups field values by series key, and then ordering is done by time.

2.6.2 Components of InfluxDB storage

The WAL (Write Ahead Log) ensures durability and stores the last offset for the storage engine to resume operation from the same offset in case of restart. [24] When the storage engine receives a write request**,** it is appended to WAL, and the data is written to disk, followed by an update to the in-memory cache. After successful confirmation, the acknowledgment is sent. The cache is an in-memory copy of data points currently in the WAL. [24] The cache organizes points by the key (measurement, tag set, and unique field), where each field is stored in its time-ordered range. [24]. The cache is queried at runtime, and the data is merged with data stored in TSM. Queries execute on the data copy in the cache during query execution. Delete operations are performed on a specific key or a time range. Like LSM (Log Structure Merge Trees), a TSM also uses a write-ahead log, read-only index files, and performs compactions to combine index files. However, it does not face deletion issues like LSMs. [29]

InfluxQL is a query language for InfluxDB similar to SQL, which provides DML and DDL statements, database management, windowing queries, aggregation function, and authentication and authorization features. [29]

CHAPTER 3

Related Work

The performance of NoSQL databases and using them as the underlying storage for big data has been subject to many studies. However, NewSQL databases have not been studied much since they have not been around for a long time, and/or most of them are proprietary solutions that have restricted open-source licenses. So, the purpose of finding related research papers was mainly focused on introspecting the research related to systems storing machine or IoT data, which is generated at a fixed interval and rapid pace and closely resembles the use case of storing monitored data in multiple databases for analysis. Since this data is enormous in volume and incredible velocity, the data falls in the big data category, and systems in contention should efficiently handle big data.

In [7], Arjun Pandya et al. evaluates the storage and performance of VoltDB and MongoDB on Industrial IoT data and concludes that VoltDB outperforms MongoDB in indexed query processing and aggregation performance due to its in-memory processing architecture. It also uses Apache Kafka as a streaming and buffer layer, helping in the data ingestion at a particular rate and batch size. An analysis was performed on the impact of batching on the two databases, and although there is a slight performance drop on VoltDB, it still performs better than MongoDB on write statistics.

In [11] compares sharded MYSQL and Google NewSQL database Spanner's insertion times for 895000 records and read performance for three different select queries and concludes that as the queries become more complex and restrictive (involving joins and limit), the performance of Spanner deteriorates, but still performs better than a sharded MYSQL database

setup. Write performance of Spanner is significantly worse than MYSQL for single node setup, but as the number of nodes increases, the MYSQL setup hits a performance bottleneck. In contrast, Spanner can scale efficiently with little degradation in performance.

[10] compares various NoSQL and distributed SQL databases by varying multiple consistency levels and replication factors across the cluster size of 1,4,6,8,12, and read and write performance were benchmarked using YCSB with varying load distribution. Cassandra outperforms MongoDB by 72.5% percent in the write-intensive workload, reiterating Cassandra is a write-intensive database. Cassandra performs significantly better with uniform distribution of load across all the nodes of the cluster. Stricter consistency levels impact the write throughput of Cassandra and VoltDB by almost 10% on an average with a constant replication factor. Read performance improves dramatically for Cassandra, VoltDB, with the introduction of replication.

[9] proposes a meteorological data storage solution using TimeScaleDB and Kafka and evaluates the performance of TimeScaleDB in comparison to Redis, MongoDB, Cassandra, and RiakTS. For scenario 1: data imported was increased by ten starting at 1000 records; for scenario 2: a restricted read query was executed on all the databases on the station_ID and duration starting from one day, one week, one month, and six months. Scenario 3 included the evaluation of indexed queries on the databases. Scenario 4 introspects performance evaluation for increasing batch size and fixed interval time on the four databases. In the end, TimescaleDB has the best and the most optimal performance in terms of resource utilization out of all the databases, followed by Cassandra, RiakTS, and MongoDB.

CHAPTER 4

Design and Implementation

Several key components are involved in developing a monitoring system, and each component was studied in great detail and which sufficed our needs were identified and finalized. The components are:

- The VoltDB cluster to be continuously monitored.
- Monitoring Agent code which retrieves the relevant monitoring data from the VoltDB cluster at varying intervals.
- Databases in the second layer, which store the data available from the kinesis layer
- Amazon Kinesis Layer to demonstrate the effect of streaming and batching.

4.1 Architecture

The VoltDB cluster (referred to as database in the first layer) has partitioning and replication to represent a practical, real-world usage scenario. We explored several workload options such as Twitter API and Yahoo! Stocks to trigger the VoltDB cluster. We chose YCSB (Yahoo! Cloud Service Benchmark) since the workloads were defined to represent the real-world scenarios of database access and usage [18]. Monitoring Agent script was run on the cluster to obtain the monitoring data at every from every instance. There are three databases in which the monitoring data will be stored (referred to as database in the second layer). The candidates are Cassandra, YugabyteDB, and InfluxDB. The Kinesis layer ensures the data is readily and continuously available both as stream and as batches since fetching data directly from the source

will cause many database connections via multiple threads and result in the system's performance degradation.

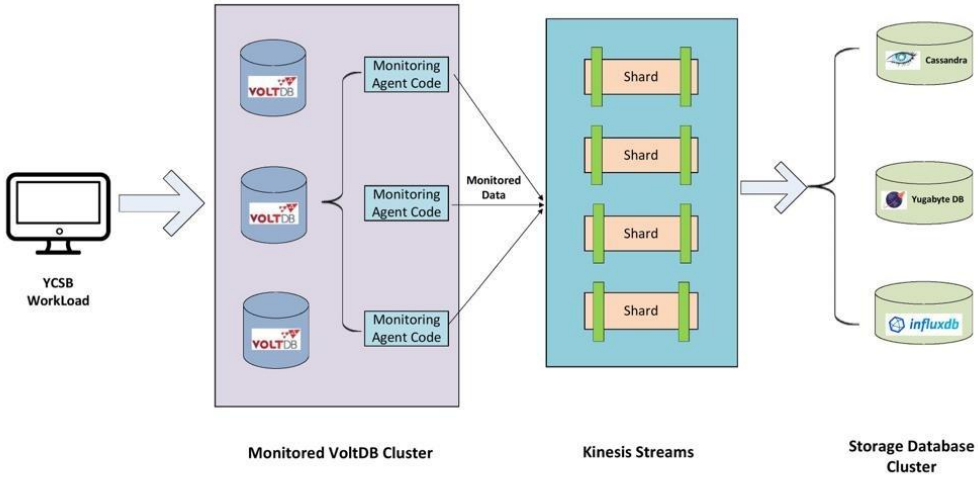

Figure 11. VoltDB Monitoring System Architecture

4.2 Monitoring Agent Code

VoltDB cluster exposes many monitoring parameters like CPU utilization, Queue Statistics via a REST API via its 'Statistics' stored procedures. These need to be collected and processed. Since many parameters will not be relevant, a Django-based API was created and hosted, which performed parameter filtering from the JSON obtained from the VoltDB API. This API was also used to write records to the Kinesis buffer for batching and streaming purposes. Every database in the second layer follows a different data model, and the data from the API was recalibrated according to each database. However, the same data was stored for each database under consideration in the second layer for similar comparison. An exhaustive study was performed to

identify the relevant monitoring parameters which need to be stored in the database in the second layer. The YugabyteDB client uses a 'prepare-bind-execute' paradigm where prepared statements are used, which can be executed repeatedly, and each time the value present value of the bind variable is evaluated and sent to the database. The PreparedStatement is not parsed again, nor is the template passed to the server again, which reduces the parsing overhead and improves write performance. Several categories of monitoring parameters were identified and categorized as follows: (Cluster Overview, Hardware Statistics, Performance Statistics, Database Events, Replication Status, Schema/Table Information).

4.2.1 Cluster Overview

Field Name/ Command	*Stored Procedure Command*	*Description*
full_cluster_size	System Information	Number of nodes in the cluster
live_nodes	System Information	Number of live nodes
dead_nodes	System Information	Number of dead nodes
database_version	System Information	Current Database version
partition_count	Partition	Total number of data partitions in each node

Table 1. Cluster Overview Parameters

4.2.2 Hardware Statistics

Field Name/ Command	*Stored Procedure Command*	*Description*
percent	CPU	Percentage of total CPU used by VoltDB process

RSS (Resident Set Size)	MEMORY	The total memory allocated to the VoltDB process on the server
java_used	MEMORY	The total memory allocated by Java and currently in use by VoltDB.
tuple_data	MEMORY	Total memory used for storing database records
index_memory	MEMORY	Total memory used for storing database indexes
string_memory	MEMORY	Total memory to store string, binary and geo-spatial data
tuplecount	MEMORY	Total number of database records in memory
new_gen_gc_count	GC	The Number of 'young generation' garbage collection
old_gen_gc_count	GC	The Number of 'old generation' garbage collection

bytes_written	IOSTATS	Total Number of bytes sent from host to client
bytes_read	IOSTATS	Total Number of bytes sent from the client to host
current_depth	QUEUE	Number of tasks waiting in queue for each host
poll_count	QUEUE	Number of tasks in executing state
avg_wait	QUEUE	The average length of time of tasks waiting in the queue

Table 2. Hardware Statistics Parameters

4.2.3 Performance Statistics

Field Name/ Command	*Stored Procedure Command*	*Description*
P99	LATENCY	The 99th percentile latency
P999	LATENCY	The 99.9th percentile latency
max	LATENCY	The maximum latency during the interval
tps	THROUGHPUT	Number of transactions per second
read	THROUGHPUT	Number of Read operations per second

insert	THROUGHPUT	Number of Insert operations per second
update	THROUGHPUT	Number of Update operations per second
delete	THROUGHPUT	Number of Delete operations per second
avg	IDLETIME	Avg. time an execution task had to wait for a new task
max	IDLETIME	Max. time an execution task had to wait for a new task
cache_hits	PLANNER	Number of queries that hit the cache
cache_misses	PLANNER	Number of queries that missed cache and had to be fetched from the disk

Table 3. Performance Statistics Parameters

4.2.4 Database Events

Field Name/ Command	Stored Procedure Command	Description
megabyte_per_second	REBALANCE	The average amount of data moved per second in MBs
percentage_moved	REBALANCE	Percentage of a total segment already moved

startup	SYSTEMINFORMATION	Time elapsed since the last startup
shutdown	SYSTEMINFORMATION	Time elapsed since any node was shutdown
nonce	SNAPSHOT	Unique ID of the snapshot
table	SNAPSHOT	Names of the tables written in the snapshot file
duration	SNAPSHOT	Time taken to complete the snapshot

Table 4. Database Events Parameters

4.2.5 Replication Stats

Field Name/ Command	*Stored Procedure Command*	*Description*
average	DRCONSUMER	The average rate of replication from
replication_rate_5M	DRCONSUMER	The average rate of replication over the past five minutes
total_bytes	DRPRODUCER	Total bytes of queued for transmissions from the replica

Table 5. Replication Statistics Parameters

4.2.6 Schema/Table Data

Field Name/ Command	*Stored Procedure Command*	*Description*

tuple_count	Table	Number of rows in each partition
tuple_allocated_memory	Table	The total memory allocated to each partition
string_data_memory	Table	The total memory allocated for storing non-inline variable-length data.
memory_estimate	Index	Memory consumed by each index entry.
procedure	Procedure	The name of the procedure
avg_execution_time	Procedure	Execution time of each procedure
invocations	Procedure	Total number of invocations of the procedure

Table 6. Schema/Table Statistics Parameters

4.3 Amazon Kinesis Layer

Amazon Kinesis is a distributed streaming and batching platform that differs from the standard queue in providing redundancy and a message bus to reach a throughput of the orders of millions of records per second. Kinesis is ideally suited for our streaming and batching needs as the data from the VoltDB is generated continuously. Furthermore, since reads and writes to the Kinesis cluster can co-occur, it helps to simulate the impact of streaming on the databases in the

second layer. Internally, Kinesis consists of a commit log, which is ordered, immutable and fault-tolerant.

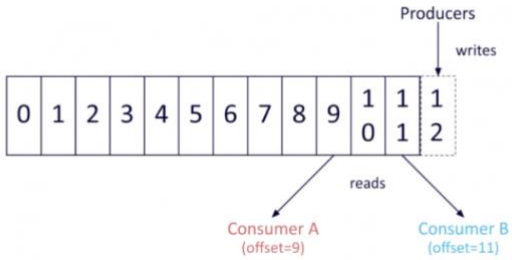

Figure 12. Log Commit in Kinesis Data Stream [16]

Unlike message queue, the same messages can be consumed by multiple consumers, which helps in the accurate analysis since the same data is written to all the databases at any instant. Each consumer (in our case, the databases in the second layer) consumes the data from each shard simultaneously. Furthermore, Kinesis also provides the functionality to replay the messages from any given offset. Thus, in case of any abruptions in data consumption in any of the consumers, the consumption of messages will resume from the same offset where it was abrupted.

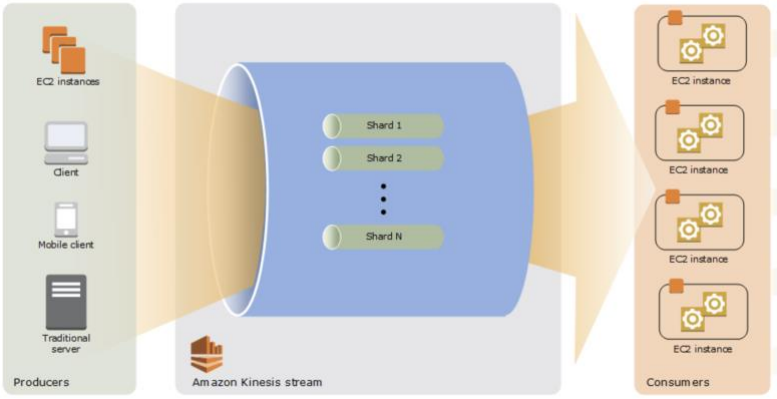

Figure 13. Kinesis Data Stream Architecture and Data Flow [13]

4.4 Data Modelling for the databases in the second layer

This section will specifically focus on the data modeling aspect of each database in the second layer. Since each database has a different data modeling and query language, single modeling might not be suitable for all the databases. The specific data model for each database is defined such that it optimizes the performance of each database.

Since the VoltDB cluster generates monitoring data at intervals of seconds or even less, it would create many partitions in each database if such granular data were to be stored. Cassandra defines the primary key during table creation and cannot be changed once the table is created. The primary key consists of a partition key, based on which the data is partitioned across the cluster, and a clustering key used to define the sorting order of the data within the partitions. Hence, Cassandra, a bucketing strategy was implemented to reduce the number of partitions needed to be accessed while querying the data. Since there are ten tables involved, each storing data for CPU statistics, memory statistics, throughput data, latency data, idle time statistics, IO stats, garbage collection data, system information, table statistics, and index information, the partition key and the clustering key is constant across the tables so that the keys are synchronized. Data at any instant is persisted with the same primary key across the keyspace. An ideal strategy is keeping the size of each partition under 100MBs of data. A compaction strategy called 'TimeWindowCompaction' is used in [17][28] with the compaction unit as 'date', which helps deal with the overhead of compacting large partitions and keeps the CPU usage and I/O under control. We leveraged a form of bucketing to break the large partitions into smaller ones. We use the host and date as the partition key and timestamp sorted in reverse order as the clustering key, ensuring that the work can be spread across the entire cluster rather than only a single node working performing much work.

YugabyteDB provides the YCQL (Yugabyte Cloud Query Language), with its roots in Cassandra Query Language [27]. Data modeling is also based on the Cassandra data model. Still, the underlying storage DocDB is a strongly consistent document store built on top of RocksDB, which impacts the performance of YugabyteDB and separates it from Cassandra. Also, the transactions are ACID compliant with strong consistency, which has a significant impact on the performance of the database.

InfluxDB is designed to fast incoming time-series data efficiently, so timestamp is the mandatory component of the data frame. InfluxDB database stores points, and every point has four components: a measurement, a tagset, a fieldset, and a timestamp. [15] The tagset is a dictionary of key-value pairs and stores data with a point. The fieldset consists of scalar values. As mentioned previously, sharding of data across the cluster of nodes is implicitly handled in InfluxDB based on the retention policy selected on the shard group for better compression and data drop.

The retention policy describes how long the data is stored in InfluxDB; the default is **autogen**, having infinite retention and replication factor 1. The most crucial concept of InfluxDB is a measurement that collects data with the standard retention policy, measurement, and tags.

Consider the following example for the line protocol output of InfluxDB:

‘**temperature,machine=unit42,type=assembly,internal=32,external=100 1434055562000000035’.** [15]

The measurement is temperature; tag set is machine = unit42, type = assembly.

The keys, machine, and type are tag keys, and 42 and assembly would be values.

Fieldset is internal = 32, external = 100. The field keys are internal and external, and field values are 32 and 100.

CHAPTER 5

Experiment and Analysis

We performed five experiments that test various aspects of the three databases in the second layer. These experiments cover a wide range of performance indicators such as the impact of streaming and batching on each database, the impact of various read queries, horizontal scalability of databases, the efficiency of indexes of the databases, and the impact of consistency and replication on the throughput of the databases.

5.1 Experiment Setup

A total of 3 clusters of 6 nodes each were set up for Cassandra, YugabyteDB, and InfluxDB were set up in AWS (Amazon Web Services) to perform the following experiment. A 3-node cluster was set up for VoltDB, which is the cluster to be monitored. Another single node was used to run the YCSB (Yahoo! Cloud Serving Benchmark) workload on the databases. This workload helped to simulate various real-time database usage scenarios. Amazon Kinesis captures a massive amount of data per second, and the data is available within millisecond enabling real-time processing; hence it will be used for efficient batching and streaming described in the first experiment. Three node clusters were used to perform the first three experiments, while experiments 4 and 5 were performed on six node clusters.

Cluster	*Service*	*Number of Nodes*	*Instance Type*
VoltDB	Amazon EC2	3	C5.2xlarge
Cassandra	Amazon EC2	6	C5.2xlarge
YugaByteDB	Amazon EC2	6	C5.2xlarge

InfluxDB	Amazon EC2	9 (3 meta and 6 data nodes)	C5.2xlarge

Table 7. Cluster Configurations

5.2 Impact of Streaming and Batching test

A batch is a collection of data points that are assembled within a time interval. [23] By definition, batching data requires all the data needed for the batch to be loaded to some storage and then processed. The latency of batch processing can be in minutes to hours [23]. In stream processing, data on a rolling window or most recent record is processed. Stream processing allows us to process data in real-time as they arrive from the source within a short time from receiving the data. [16]

This experiment was performed to evaluate the impact of batching and streaming on each database in the second layer. An API program was developed to create the batch of appropriate size records and push them into the candidate databases. Similarly, a program was written to read the messages from the Kinesis stream to databases in the second layer in a streaming manner. A total of 100,000 transactions with each batch size of 10000 transactions was executed. Similarly, for a fair comparison, a stream of 100,000 writes was performed on each database. Here, the performance evaluation metric is the execution time; the lower, the better. Each experiment was performed three times, and an average is presented.

5.2.1 Results

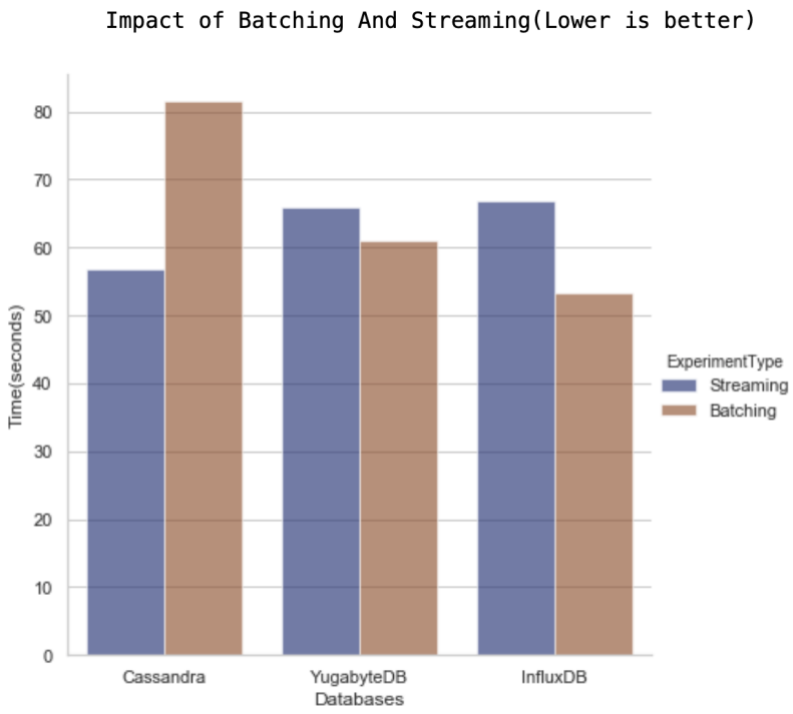

Figure 14. Impact of Streaming and Batching

5.2.2 Analysis

The impact of batching and streaming on the three databases are as follows:

- Cassandra provides excellent write performance with a stream of data. The performance is due to Cassandra's multi-master architecture, where every node can accept read/write requests parallelly. There is no bottleneck of master-slave replica data synchronization, thus increasing the overall throughput of the cluster. Also, high write performance is due to the way Cassandra handles writes in an append mode.

Cassandra's performance degrades as batching is introduced. In general, batching is discouraged in Cassandra and is only recommended in cases of atomic transactions. Due to multi-master architecture, the batch write request may be accepted by any node in the cluster, and subsequently, the node redirects the write request to the appropriate node with the appropriate partition. Due to this, the coordinator node has to do a lot more work than any other node, which causes a decrease in throughput. Cassandra also warns of any batch size more than 5KB, and the performance degrades as each of the transaction insertions in our case was 16KB in size.

- In YugabyteDB, the write requests issued are first handled by the query layer, which is translated into an internal key and appropriate tablet (partition). An RPC call is made to the YB-Master to identify the YB-TServer holding the key. Since YugabyteDB provides serializable isolation and strong ACID guarantees, it creates an overhead: YQL finding the specific YB-TServer, acquiring a lock on the Raft leader, and performing the update. The Raft followers then acquire an update log from the leader before an acknowledgment is sent to the leader.

  Batch operations in YugabyteDB allow the user to send multiple write operations in one RPC call. Although the latency of batch transactions might be higher than the single operations, the throughput is higher due to fewer round-trips from the application to the database. Using the 'Prepare-bind-execute' paradigm with client code instead of inlining the literals also resulted in eliminating statement reparsing overhead and reducing execution time.

- InfluxDB employs different compression algorithms for different data types due to the granularity of the data with which it deals. While this results in efficient memory and disk space utilization and helps drop the data beyond the retention policy, it consumes resources allocated for handling incoming requests. Although InfluxDB performs well with our data which is very similar to a time-series data stream, there is a little performance degradation due to aggressive compression running in the background depending on the datatype selected, but the performance is still comparable to YugabyteDB on stream data.

  Batching in InfluxDB collects data points using the InfluxDB line protocol format. Batching in InfluxDB has much better performance due to a reduction in the number of HTTP requests to the database since multiple batches of points are being submitted to the database in a single HTTP request. InfluxDB recommends a batch size of around 5000-10000 data points for optimal performance.

  Conclusion: Cassandra performs best with streaming data, but the performance of InfluxDB and YugabyteDB improves with the introduction of batching.

5.3 Query Performance Test

Four queries were selected to test each database. Each query is a representative use case for a monitoring system. We executed each query fifteen times and calculated the average of the execution times. We dropped outliers in our computation since they might have been due to network interference. The queries were executed on non-indexed columns on a 3-node cluster across all the databases with replication factor two (Quorum). Each database houses approximately 7.5 GB of monitored data, with some variation due to running background compactions. Query 1

is without any predicate. Queries 2 and 3 come with more restricted predicates. Query 4 is an aggregate query within a time interval. The query functions are:

Query 1: All the data from all the nodes

Query 2: All the data for a specific node

Query 3: All the data for a specific node within a specific time interval

Query 4: Maximum CPU utilization between a specific time interval

5.3.1 Results

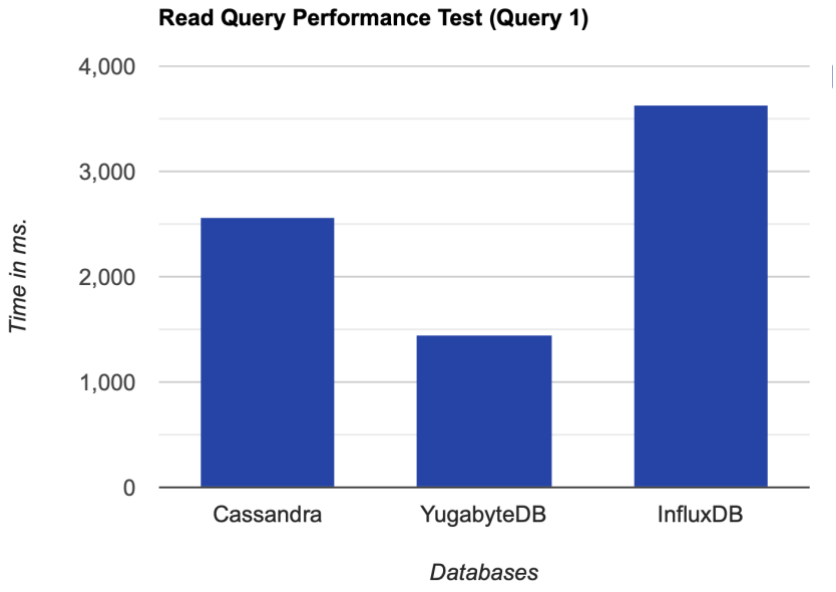

Figure 15. Query 1 Performance

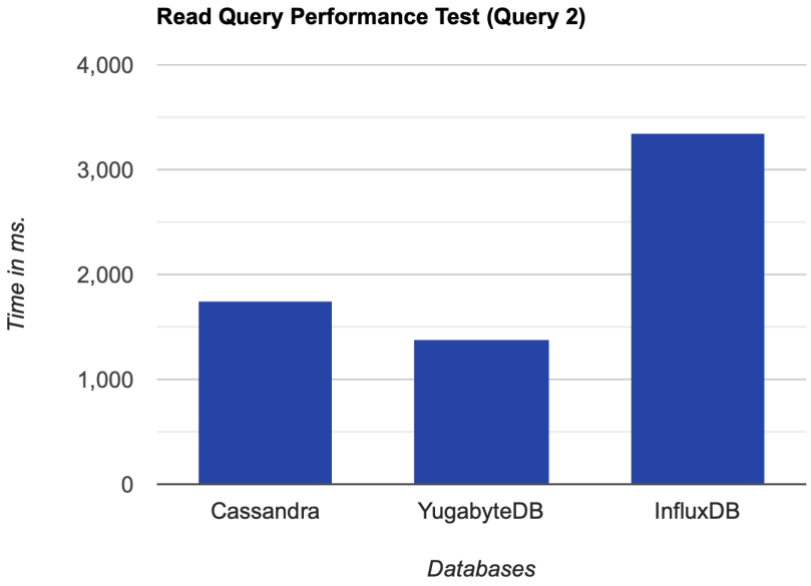

Figure 16. Query 2 Performance

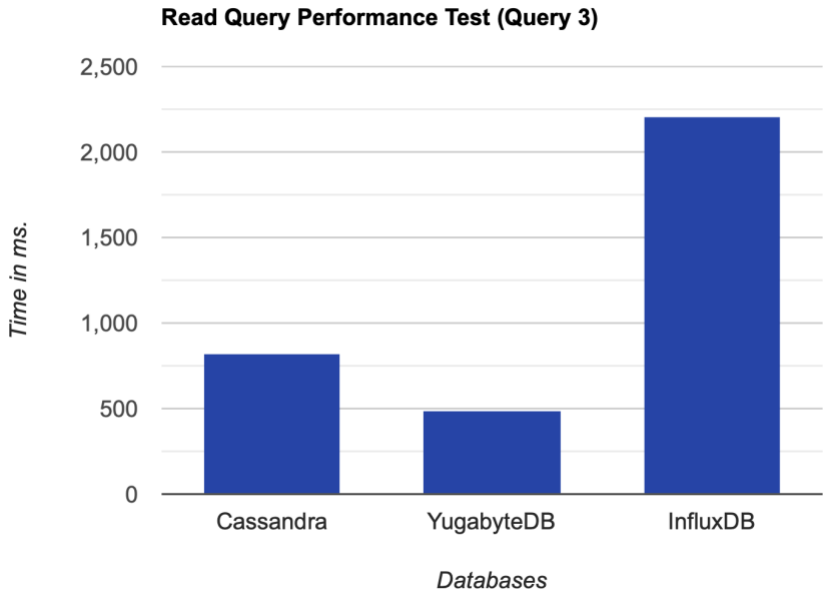

Figure 17. Query 3 Performance

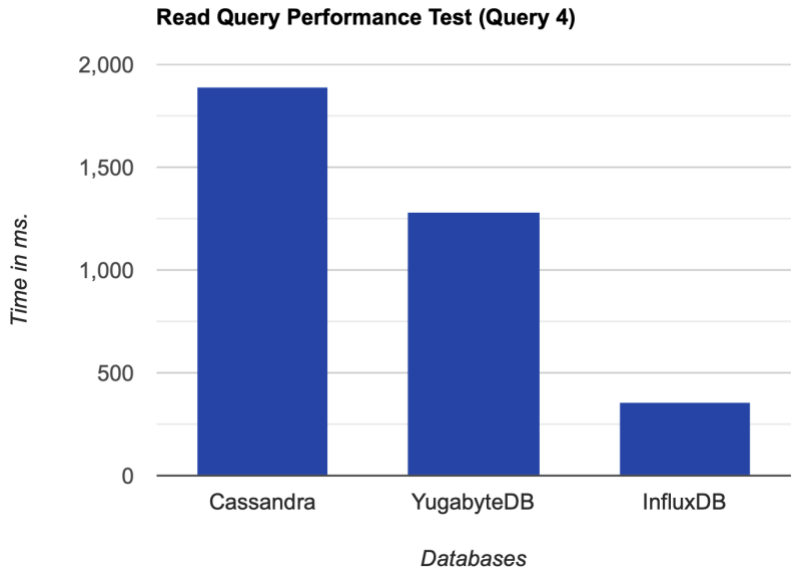

Figure 18. Query 4 Performance

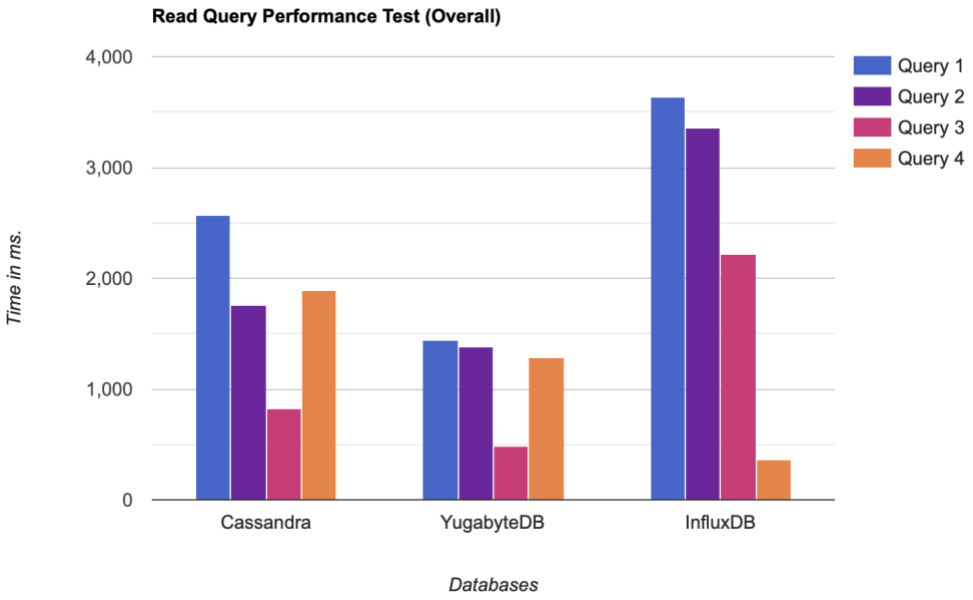

Figure 19. Overall Read Query Performance

5.3.2 Analysis

The analysis of read query performance is as follows:

- Since the read queries require broadcasting operations that probe the entire cluster, Cassandra's performance improves according to the restrictions imposed on the queries from query1 to query2 and from query 2 to query 3. This is because the targeted operations can perform better when the predicates include the primary key. In query 2, the performance is better than query 1 due to the introduction of a predicate on host_id, but since the primary key is on host_id and date, many partitions have to be scanned to get the results. On the other hand, with query 3, the appropriate time interval is identified along with host_id; hence fewer partitions are scanned, and performance improves dramatically. Scanning all the partitions requires a substantial amount of time, making the performance worse for queries 1 and 4.
- Since YugabyteDB uses the RAFT consensus protocol, data with the quorum leader is strongly consistent. So, read operations require only a single read from the leader without probing other replicas on other nodes. As for Cassandra, for strongly consistent reads, a quorum is required. Hence, YugabyteDB is a highly read performant database. The internal design of DocDB, which is the strongly consistent distributed document store of YugabyteDB, allows it to keep min/max and metadata values of the clustered columns (timestamp in our case) and store that in SSTable as metadata. Thus, query 3 is highly optimized in YugabyteDB as the number of SSTables to scan to find the required data is minimized.
- The time series underlying data format (Time Structured Merge Tree) is columnar as opposed to LSM based databases which are row-oriented. Hence, columnar databases are

expected to be hit with columnar queries (select column_name from) instead of (select * from) in row-oriented databases. Hence, InfluxDB performs worse for most row-oriented queries. Query 3 performs better as filtering by time is optimized for time-series databases. Since the older data is less critical in time-series databases, the queries dealing with snippets of time with recent timestamps will be more efficient. Also, due to the column-oriented architecture, aggregations are much more efficient to perform within a time window.

Conclusion: Cassandra's performance improves due to restrictions on the query, YugabyteDB performs the best on read operations overall, and InfluxDB performs best on windowing aggregation read queries.

5.4 Efficiency of Index Test

Secondary Indexes were created on each database. We increased the size of the databases gradually from 2.5GBs to 25GBs to study the efficiency of the indexing mechanism of each database for the given database size. Since secondary indexes are not recommended for columns with too high or too low cardinality, we created an index on the CPU usage column, of which domain has integer values between 0 and 100. Ten iterations of each query were executed, and an average of the values was considered, and outliers were ignored since there might be network interference in some executions. Queries were executed to fetch all the data when the value of CPU utilization went above eighty-five percent since this is a typical use case to detect anomaly in the monitoring systems and did not include any primary column in the predicate.

5.4.1 Results

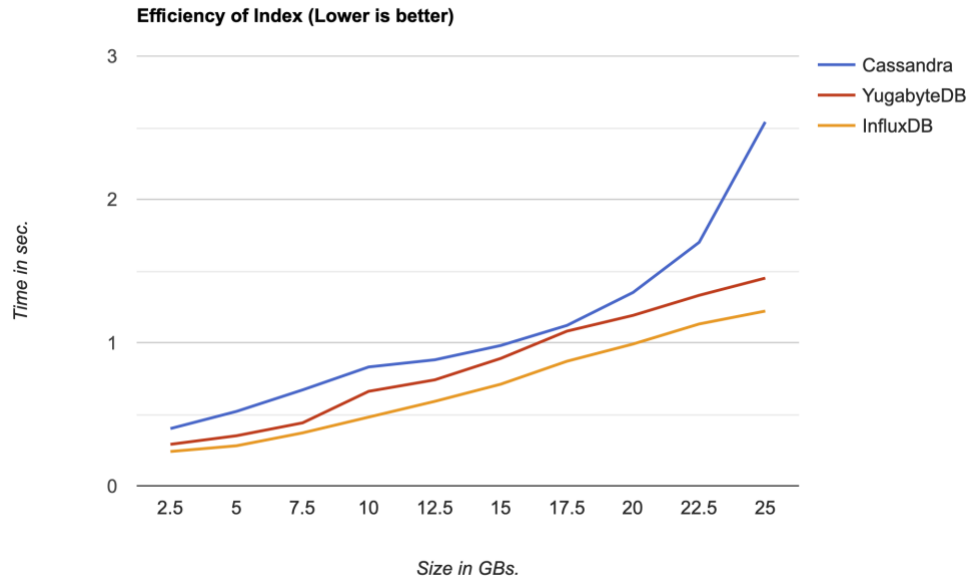

Figure 20. Efficiency of Index

5.4.2 Analysis

The impact of Indexing on the three databases is as follows:

- In Cassandra, secondary indexes are local, and hence an index is built on each shard. A larger number of records have to be scanned in each node as the data size increases, and thus, index performance keeps on decreasing. Since the data is partitioned across the nodes based on the hash value of the partition key, Cassandra nodes are aware of the positioning of the data blocks within the cluster. However, if a query predicate does not come with the partition key value, Cassandra performs broadcasting operations by probing all the nodes in the cluster to answer the query.
- In YugabyteDB, every secondary index is an internal table; the index table is internally shared and distributed across the nodes, similar to using tables. Whenever data is inserted

or updated, YugabyteDB uses distributed transactions to update both the primary and secondary index tables making the secondary index scans in Yugabytedb strongly consistent and a single targeted operation rather than reads from all the nodes/shards in the cluster, and hence YugabyteDB index is linearly scalable.

- Indexing in InfluxDB is via an in-memory index shared across the shards and provides indexed access to measurement, tags, and series. Indexing in a column can be achieved by putting it in a tag instead of in a field. There was a significant improvement in read performance when the column was put inside a tag instead of a field.

  Conclusion: InfluxDB performs best with indexing due to its in-memory indexing, followed by YugabyteDB and then Cassandra

5.5 Horizontal Scalability Test

Horizontal scalability is defined as the ability to scale the system by increasing the number of nodes in the cluster instead of vertical scaling, which requires the physical configuration of adding hardware components like CPU, Memory, and Disk in case of increased requests.

1, 3, and 6 node clusters were used to perform this experiment for each of the three databases. InfluxDB requires setting up of meta-nodes and data-nodes separately. Meta-nodes store the information about the nodes in the cluster and their role, databases, retention policies, shard and their groups, and their position in the cluster. Data nodes store the actual tag keys and values, measurement, and field keys and values. InfluxDB recommends adding an odd number of meta-nodes since it allows the meta-nodes to reach a quorum, and the number of data nodes should be divisible by the replication factor.

The experiment was performed by saturating each cluster and subsequently increasing the number of nodes in the cluster to check the increase in the throughput of the database cluster. Cassandra can be horizontally scaled by adding more nodes in the cluster and setting the same seed nodes, allowing the rebalancing of the data across each node in the cluster. Replication was enabled across each cluster since high availability becomes necessary in a distributed database. In this case, a replication factor of Quorum was considered (two for three-node cluster and four for six node cluster), wherein two copies of the data will be stored in the three-node database cluster, and four replicas will be stored in six node cluster.

Since there is a limitation on increasing the arrival rate from the Kinesis cluster to the databases capped at 4 Mb/s, we implemented multi-threading to demonstrate multiple parallel requests on each database. We stress the database resources until a point where each node in the cluster reaches a saturation point (we checked for the CPU utilization of each node to reach 85% and the average latency to reach a value of seconds (typical values are in milliseconds), which indicated that the database had reached saturation in terms of the number of requests it can process and increasing the number of requests further does not increase throughput. An average of three executions and outliers were ignored as they might have been caused due to network interference.

5.5.1 Results

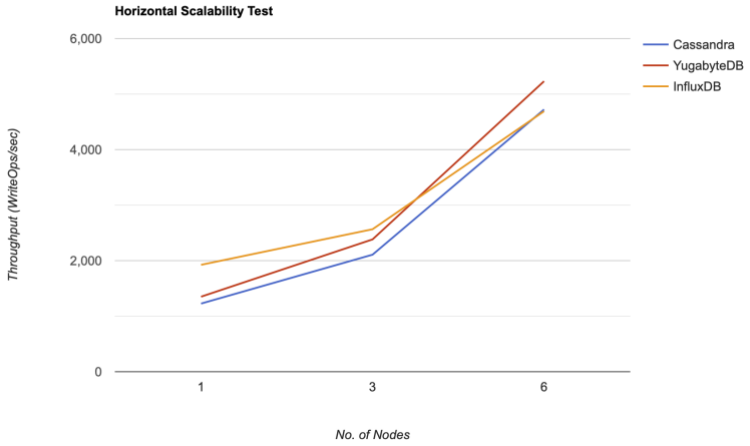

Figure 21. Horizontal Scalability test

5.5.2 Analysis

Horizontal scalability performance is as follows:

- In Cassandra, background anti-entropy maintenance is always running. Anti-entropy repair compares the data in all the replicas and repairs the inconsistent data in the cluster. The anti-entropy mechanism is an expensive operation that causes a lot of CPU utilization along with inter-node communication using gossip protocol to share the current state of the replicas. Hence, even though the Cassandra scales linearly in adding extra nodes, the issues mentioned above are amplified as the cluster size increases and cause a degradation in the performance and throughput of the database.
- As the size of the InfluxDB cluster increases, the number of meta-nodes and the data-nodes is proportionally increased. Increasing the number of meta-nodes increases inter-node communication exponentially. Each meta-node communicates to every other meta-node, and data nodes also communicate with meta-node and data-node. Thus, the horizontal scalability of InfluxDB is slightly affected by the inter-node communication overhead

between the meta-node and data-node cluster to exchange and rebalance data across the nodes.

- In YugabyteDB, when a node is added, the newly added node becomes a leader, and then the YB-Master node initiates rebalancing the follower since it manages the shard meta-data and position of each tablet leader. The moves are undertaken so that none of the existing nodes bears the burden of populating the new node. Since the Raft leader is strongly consistent for each leader, no anti-entropy maintenance is required before reading. However, YugabyteDB supports 'Serializable' isolation, which creates an overhead due to locking mechanism to perform write operations and prevent dirty reads. This effect is amplified as the number of parallel write requests on the database increases.

  Conclusion: InfluxDB performs well with a single node, but the performance gains are not proportional to the increase in the number of nodes; YugabyteDB and Cassandra almost scale linearly, with YugabyteDB outperforming Cassandra with six node cluster.

5.6 Impact of Consistency and Replication

Consistency ensures that every read request to the database receives the most recent writes [19]. Cassandra and InfluxDB offer tunable consistency, whereas YugabyteDB offers strong consistency via RAFT consensus protocol. *Replication* is defined as keeping the redundant copies of each shard across a group of nodes based on the replication factor defined. Since YugabyteDB offers strong consistency, we set the consistency level for Cassandra and InfluxDB to ALL. (Strongest Consistency Level). Consistency levels offered by Cassandra and Influx are ONE, QUORUM, and ALL.

We considered six node clusters for each of the databases, and the replication factor was increased from 1 (ONE) to 4 (QUORUM) to 6 (ALL) to demonstrate the write throughput. An average of three values was considered for each replication factor for each database. A total of 60,000 insert transactions were executed on each database, and an average of two values for each replication factor was taken.

5.6.1 Results

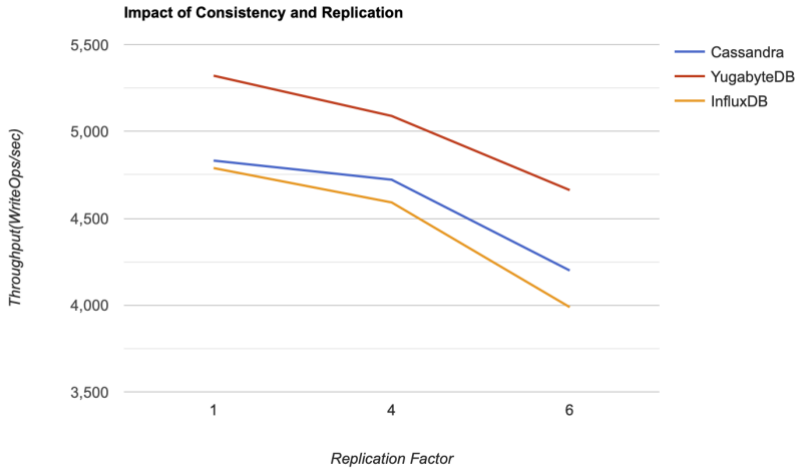

Figure 22. Impact of consistency levels and replication factors

5.6.2 Analysis

The impact of Replication factor change on each database is as follows:

- As the replication factor increases, the write performance of Cassandra and InfluxDB decreases more steeply than YugabyteDB. All the nodes having the replica have to be written to before an acknowledgment can be sent for a successful write operation. In general, Cassandra and InfluxDB are eventually consistent databases that prefer

Availability over Consistency in case of a failure and warn against the use of the strongest consistency due to performance degradation.

- The underlying storage DocDB of YugabyteDB synchronously and automatically replicates data and maintains data consistency using the raft consensus protocol. Since the underlying architecture for YugabyteDB is optimized to perform strong, consistent writes to all the nodes, it does not degrade significantly. However, only a minimum of three replicas is required for consensus. Hence, there is a slight drop in the performance of YugbyteDB beyond three replicas.

  Conclusion: YugabyteDB shows the best performance as the replication factor is increased. Cassandra and InfluxDB experience performance degradation as the replication factor is increased as both of them are eventually consistent databases.

CHAPTER 6

Conclusion and Future Work

In this project, we built a monitoring system for VoltDB, a NewSQL database, and evaluated the performance of Cassandra, YugabyteDB, and InfluxDB as candidate databases in the second layer. It was ensured that each database was modeled to perform efficiently and comparisons in each of the tests were fair. Various tests were performed, and the performance of each database was evaluated with rationale and plausible explanations.

Cassandra performs the best in write operations in a streaming manner due to its peer-to-peer architecture and its focus on availability instead of consistency in the CAP spectrum. InfluxDB performs significantly better with batching due to the reduction in the HTTP requests reducing overhead. Although all the databases suffer from issues concerning horizontal scalability, InfluxDB provides the best performance with a single node, but YugabyteDB scales the best, followed by Cassandra and then InfluxDB, due to inter-node communication overhead in InfluxDB owing to its complex architecture. We can conclude that YugabyteDB performed best on the read operations of non-indexed queries, which can be attributed to the strong consistency of data it provides due to the ACID guarantees, making read operations efficient. InfluxDB performs the best on indexed queries due to its in-memory shared indexing with the usage of tags.

The project's purpose was to evaluate candidate databases for the second layer; hence a monitoring UI is not set up. However, monitoring tools like Prometheus and Grafana are available, which can be used to monitor the clusters of the databases in the second layer, consume the data from the tables in each of the databases, and display it on a UI in a streaming manner. In the future, various other experiments can also be performed on these databases depending on the use cases.

REFERENCES

[1] 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.

[2] Joshi, Janaki, Chodisetty, Lakshmi, and Raveendran, Varsha. “A Quality Attribute-based Evaluation of Time-series Databases for Edge-centric Architectures.” Proceedings of the International Conference on Omni-layer Intelligent Systems (2019): 98-103. Web.

[3] Guy Harrison. Next Generation Databases: NoSQL, NewSQL, and Big Data. New York: Apress, 2016. Web.

[4] http://sites.computer.org/debull/A13june/VoltDB1.pdf [Online]

[5] Grolinger, Katarina, Higashino, Wilson A, Tiwari, Abhinav, and Capretz, Miriam AM. “Data Management in Cloud Environments: NoSQL and NewSQL Data Stores.” Journal of Cloud Computing : Advances, Systems and Applications 2.1 (2013): 1-24. Web.

[6] Barata, Melyssa, and Bernardino, Jorge. “Cassandra’s Performance and Scalability Evaluation.” DATA 2016 - Proceedings of the 5th International Conference on Data Management Technologies and Applications (2016): 127-34. Web.

[7] Pandya, Arjun, Kulkarni, Chaitanya, Mali, Kunal, and Wang, Jianwu. “An Open Source Cloud-Based NoSQL and NewSQL Database Benchmarking Platform for IoT Data.” Benchmarking, Measuring, and Optimizing. Cham: Springer International, 2019. 65-77. Lecture Notes in Computer Science. Web.

[8] https://docs.voltdb.com/UsingVoltDB/ [Online]

[9] Shen, Liqun, Lou, Yuansheng, Chen, Yong, Lu, Ming, and Ye, Feng. “Meteorological Sensor Data Storage Mechanism Based on TimescaleDB and Kafka.” Data Science. Singapore: Springer Singapore, 2019. 137-47. Communications in Computer and Information Science. Web.

[10] “Benchmarking Replication in NoSQL databases”
https://www.doc.ic.ac.uk/teaching/distinguished-projects/2014/g.haughian.pdf [Online].

[11] Murazzo, María, Gómez, Pablo, Rodríguez, Nelson, and Medel, Diego. “Database NewSQL Performance Evaluation for Big Data in the Public Cloud.” Cloud Computing and Big Data. Cham: Springer International, 2019. 110-21. Communications in Computer and Information Science. Web.

[12] Gorbenko, Anatoliy, Romanovsky, Alexander, and Tarasyuk, Olga. “Interplaying Cassandra NoSQL Consistency and Performance: A Benchmarking Approach.” Dependable Computing - EDCC 2020 Workshops. Cham: Springer International, 2020. 168-84. Communications in Computer and Information Science. Web.

[13] 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].

[14] “Suitability of Influxdb Database for Iot Applications.” International Journal of Innovative Technology and Exploring Engineering 8.10 (2019): 1850-857. Web.

[15] https://www.influxdata.com/blog/influxdb-internals-101-part-one/ [Online]

[16] 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.

[17] 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.

[18] 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.

[19] Hewitt, Eben. Cassandra: The Definitive Guide. First ed. Beijing ; Sebastopol, California: O’Reilly, 2011. Web.

[20] https://docs.yugabyte.com/latest/architecture/layered-architecture/ [Online]

[21] https://docs.yugabyte.com/latest/architecture/docdb-sharding/sharding/ [Online]

[22] https://docs.yugabyte.com/latest/architecture/docdb/persistence/ [Online]

[23] https://thenewstack.io/the-big-data-debate-batch-processing-vs-streaming-processing [Online]

[24] https://docs.influxdata.com/influxdb/v1.8/concepts/ [Online]

[25] https://docs.influxdata.com/influxdb/v1.8/concepts/clustering/#optimal-server-counts [Online]

[26] Pavlo, Andrew, and Aslett, Matthew. “What’s Really New with NewSQL?” SIGMOD Record 45.2 (2016): 45-55. Web.

[27] https://www.voltdb.com/wp-content/uploads/2017/05/VoltDB_SQL-vs-NoSQL-vs-NewSQL.pdf [Online]

[28] “Cassandra for Internet of Things: An Experimental Evaluation” https://www.scitepress.org/Papers/2016/58464/58464.pdf