Project Report

An Empirical Evaluation of Distributed Key/Value Storage Systems

CS550 – Advanced Operating System

November 29th 2017

Adithya Sreenath Chandrashekarapuram – A20402135

Krishna Bharadwaj – A20398222

Contents

1. Introduction

Distributed databases are usually non-relational databases that make a quick access to data over a large number of nodes possible. These databases can be used as key/value storage systems that allow for storing and accessing data by usage of key-value pairs. The latency, the time per operation taken from a request to be submitted from a client to a response to be received by the client and throughput of a system, which is the number of operations a system can handle over some period of time play a major role while looking for an optimal storage system.

Applications have become more demanding in the present scenario with regards to data being fetched in real- time. This data many a times would be unstructured. This leads to the requirement of databases that need to be extremely agile in dealing with the unstructured data and deliver to the applications in real-time. This has led to the birth of many NOSQL databases which have proven to perform and scale better than our traditional relational databases.

1.1 Redis

Redis is an open-source in-memory database project implementing a distributed, in-memory key-value store with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, hyperloglogs, bitmaps and spatial indexes.

The name Redis means **RE**mote **DI**ctionary **S**erver. Redis maps keys to types of values. An important difference between Redis and other structured storage systems is that Redis supports not only strings, but also abstract data types. The type of a value determines what operations (called commands) are available for the value itself. Redis supports high-level, atomic, server-side operations like intersection, union, and difference between sets and sorting of lists, sets and sorted sets.

Redis supports master-slave replication. Data from any Redis server can replicate to any number of slaves. A slave may be a master to another slave. This allows Redis to implement a single-rooted

replication tree. Redis slaves can be configured to accept writes, permitting intentional and unintentional inconsistency between instances.

When the durability of data is not needed, the in-memory nature of Redis allows it to perform well compared to database systems that write every change to disk before considering a transaction committed.[5] Redis operates as a single process and is single-threaded or double-threaded when it rewrites the AOF (append-only file).

In summary Redis has the following features:

- No steep learning curve
- Highly generic.
- Can be used to solve many problems
- In-memory key-value data store, thus performing operations very fast
- Supports multiple data types
- Supports scalability

1.2 MongoDB

MongoDB is a free and open-source cross-platform, document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas. MongoDB is developed by MongoDB Inc.

MongoDB supports field, range queries, regular expression searches. Queries can return specific fields of documents and also include user-defined JavaScript functions. Queries can also be configured to return a random sample of results of a given size.

Fields in a MongoDB document can be indexed with primary and secondary indices. MongoDB provides high availability with replica sets. A replica set consists of two or more copies of the data. Each replica set member may act in the role of primary or secondary replica at any time. All writes and reads are done on the primary replica by

default. Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically conducts an election process to determine which secondary should become the primary. Secondaries can optionally serve read operations, but that data is only eventually consistent by default.

MongoDB scales horizontally using sharding. The user chooses a shard key, which determines how the data in a collection will be distributed. The data is split into ranges (based on the shard key) and distributed across multiple shards. (A shard is a master with one or more slaves.). Alternatively, the shard key can be hashed to map to a shard – enabling an even data distribution. MongoDB can run over multiple servers, balancing the load or duplicating data to keep the system up and running in case of hardware failure.

In summary MongoDB has the following features:

- Documents based key-value store.
- Supports dynamic queries.
- Easy scalability.
- Need to measure the speed at which insertions, deletions and data retrievals happen

1.3 Cassandra

Apache Cassandra is a free and open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacentres with asynchronous master less replication allowing low latency operations for all clients.

Apache Cassandra has the following features:

- **Decentralized**
  Every node in the cluster has the same role. There is no single point of failure. Data is distributed across the cluster (so each node contains different data), but there is no master as every node can service any request.

- **Supports replication and multi data centre replication**
  Replication strategies are configurable. Cassandra is designed as a distributed system, for deployment of large numbers of nodes across multiple data centers. Key features of Cassandra's distributed architecture are specifically tailored for multiple-data centre deployment, for redundancy, for failover and disaster recovery.
- **Scalability**
  Designed to have read and write throughput both increase linearly as new machines are added, with the aim of no downtime or interruption to applications.
- **Fault-tolerant**
  Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.
- **Tunable consistency**
  Writes and reads offer a tunable level of consistency, all the way from "writes never fail" to block for all replicas to be readable", with the quorum level in the middle.
- **Map Reduce support**
  Cassandra has Hadoop integration, with Map Reduce support. There is support also for Apache Pig and Apache Hive.

2 Problem Statement

There is a growing need to evaluate NoSQL databases due to their immense popularity in the last few years. Choosing the right storage system can become a challenging task due to the presence of numerous options. Evaluation of the different factors that affect the key/value storage systems is required when selecting one for a particular purpose. The latency and throughput of multiple systems can be compared by sending all of the key-value pairs through a client API for insert, then lookup, and then remove. Plotting of graphs to depict these comparisons will allow for a user to pick an optimal systems for his purpose.

3 Related Work

Benchmarking has been performed for Cassandra *by Adrian Cockcroft and Denis Sheahan.* Netflix is currently using Cassandra for its globally distributed streaming product. They came to the conclusion that Cassandra scales linearly very efficiently and its very rapid deployment automation makes it easy to manage.

Link: https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

Benchmarking MongoDB has been done by Percona's team. The database tests are implemented in Lua scripts, and the benchmarking is done via sysbench

Link: https://www.percona.com/blog/2013/03/14/sysbench-benchmark-for-mongodb/

Benchmarking Redis has been performed by Redis Labs. They explain the performance metrics of Redis and explain the 5 key metrics for which performance depends upon.

Link: https://blog.newrelic.com/2015/05/11/redis-performance-metrics/

4 Proposed solution

For this project we have chosen the following databases to make our evaluations –

- MongoDB
- Cassandra
- Redis

We set up these databases in a distributed environment and ran some pre prepared code stubs for insert, select and delete operations to calculate the throughput and latency using Yahoo's YCSB cloud benchmarking tool. With this tool, we were able to accurately measure the required outputs and create graphical plots for each of the databases, varying parameters such as the workload and the number of clusters.

5 Setup and Environment

The steps involved in the setup are as follows:

Creation of the **EC2** instances in AWS for a distributed environment

We first created 3 Ubuntu 16.04 instances with the following configuration:

- T2.micro
- 1GB RAM
- Single Core
- 30GB storage

5.1 MongoDB

Installation and setup of 3 node cluster of MongoDB version 3.0.

Sharding was enabled by running the following commands in the Mongo Client:

sh.enableSharding("ycsb")

sh.shardCollection("ycsb.usertable", { "_id": 1})

5.2 Cassandra

Installation and setup of 3 node cluster of Cassandra version 2.1

Created a keyspace and table for YCSB benchmark from the Cassandra client (CQLSH)

5.3 Redis

Installation and setup of 3 node cluster of Redis 3.0.0

No addition steps were required for YCSB benchmark.

5.4 YCSB

Setup YCSB and its dependencies from the following source:

https://github.com/brianfrankcooper/YCSB

6 Experiment

We ran the predefined workload A and workload B on 3 nodes for each of the databases:

6.1 Workload A

This workload consists of ratio of 50:50 Read/Write operations respectively. Each key-value pair is of the size 1Kb in total. We ran 100,000 operations which in effect means that 50,000 Kb of data was stored/retrieved.

6.2 Workload B

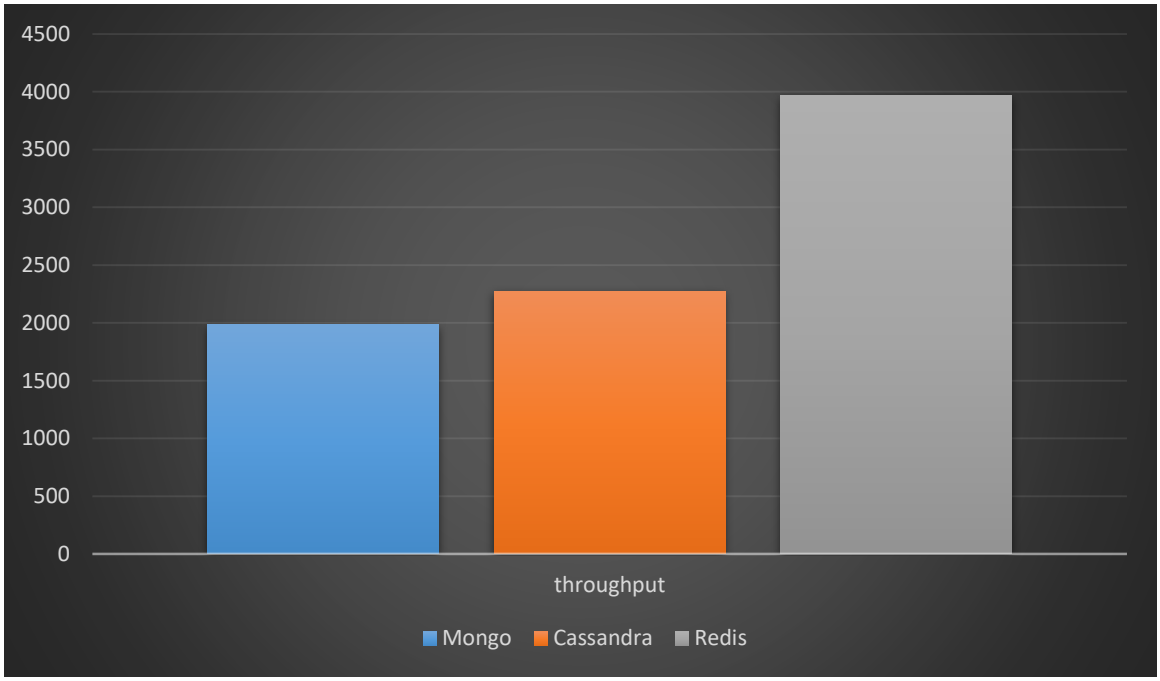
This workload is a read heavy workload which runs a ratio of 95:5 Read/Write operations respectively. Each key-value pair is of the size 1Kb in total. We ran 100,000 operations which in effect means that 50,000 Kb of data was stored/retrieved.

Upon running these workloads we calculate the throughput and latency for each setup.

7 Results and Analysis

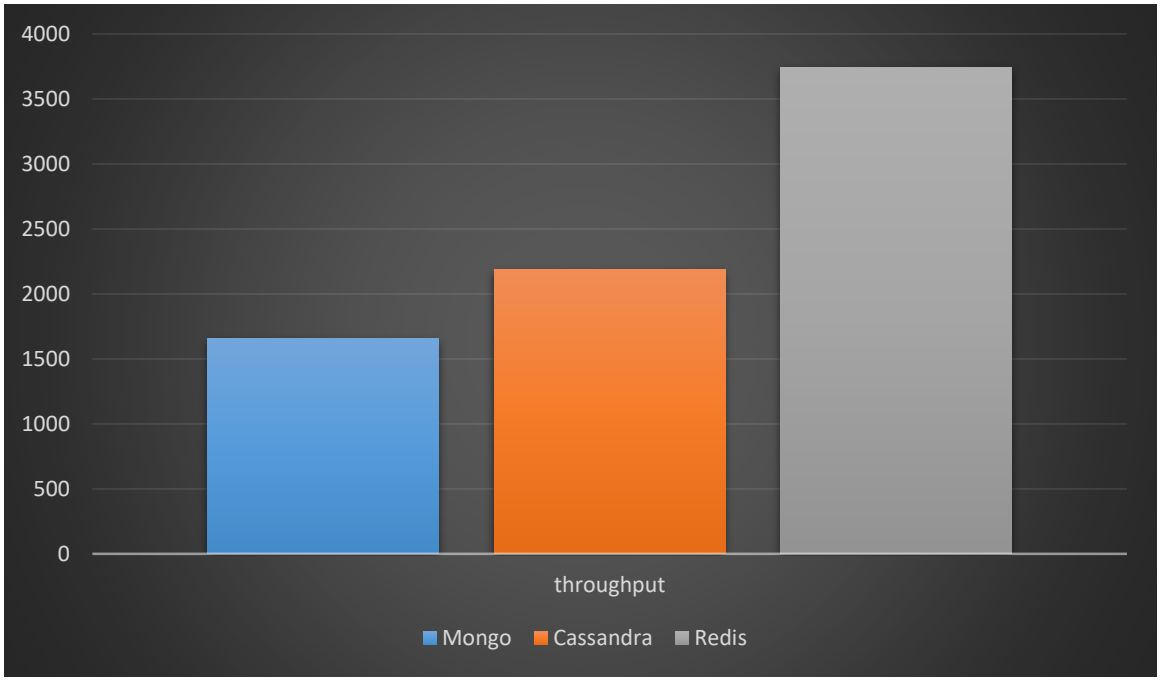
7.1 Average Throughput

For 100,000 operations



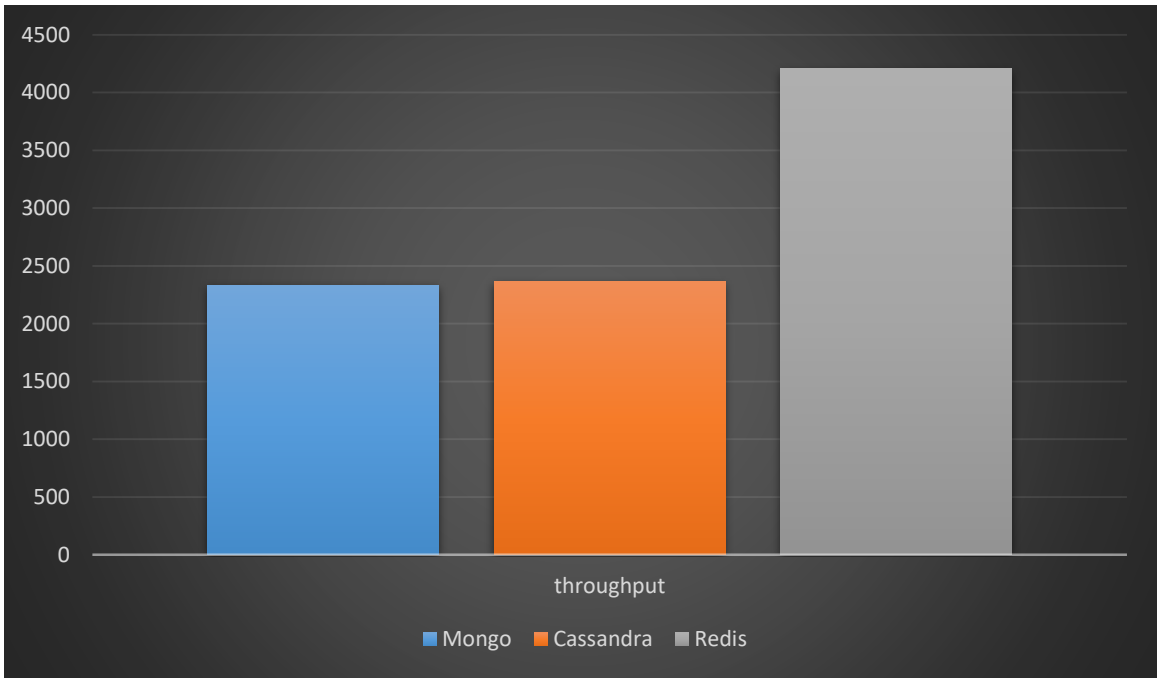
7.2 Workload A

For 100,000 operations



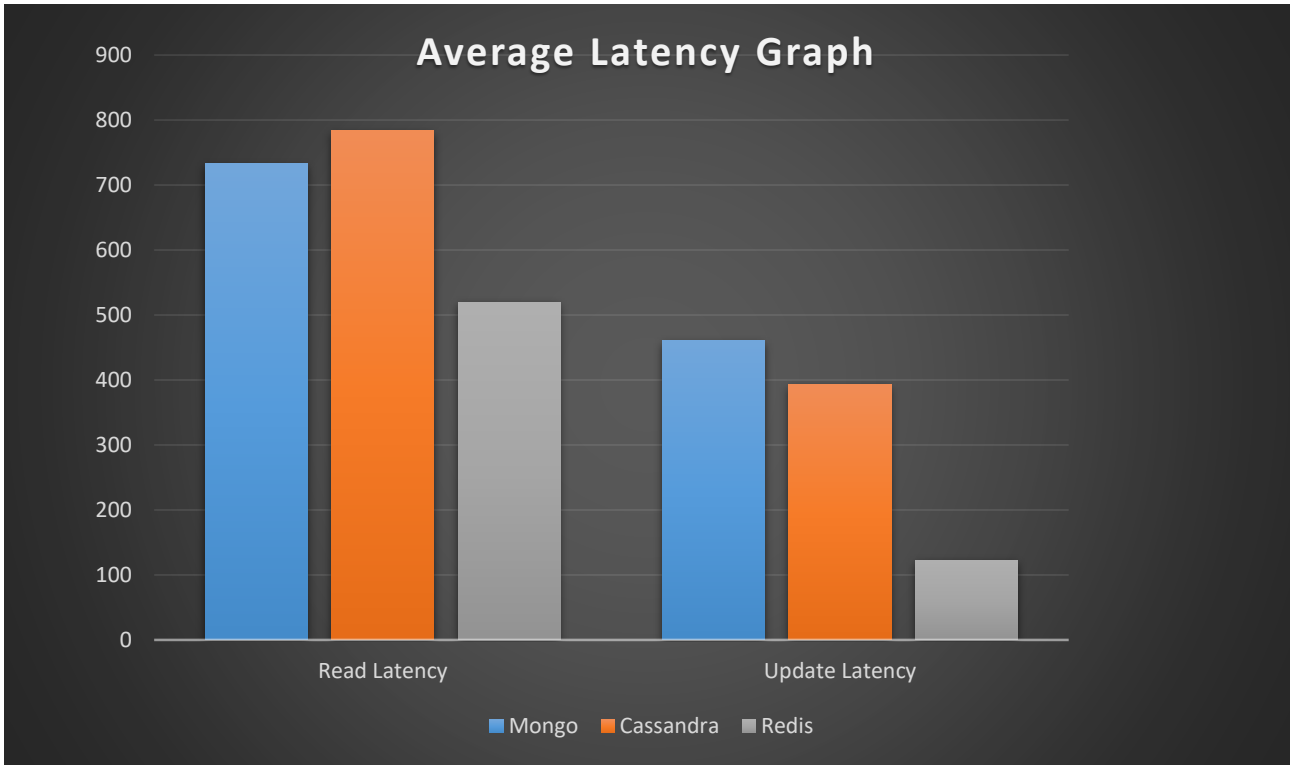
7.3 Workload B

For 100,000 operations



7.4 Average Latency

For 100,000 operations



8 Conclusion

As we can see from the depictions above, Redis consistently perfors metter than MongoDB and Cassandra when it comes to parameters of throughput and latency. MongoDB and Cassandra have similar Throughput as we have tested based on a single thread. Based on Cassandras configuration we can assume that Cassandras throughput increases as the number of threads increases as it exploits parallelism.

9 Work Distribution

Adithya Sreenath

- Research on databases and gathering information on how to go about the project
- Setup of Cassandra
- Setup of MongoDB
- Benchmarking Cassandra
- Result Analysis of Redis
- Result Analysis of MongoDB
- Documentation

Krishna Bharadwaj

- Research on databases and gathering information on how to go about the project
- Setup of Redis
- Benchmarking Redis
- Benchmarking MongoDB
- Result Analysis of Cassandra
- Documentation

10 References

1. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark
2. http://cassandra.apache.org/
3. https://www.mongodb.com/
4. https://redis.io/
5. http://www.aerospike.com/what-is-a-key-value-store/
6. https://en.wikipedia.org/wiki/Apache_Cassandra
7. https://en.wikipedia.org/wiki/MongoDB
8. https://en.wikipedia.org/wiki/Redis
9. https://github.com/brianfrankcooper/YCSB/wiki