CS553 Cloud Computing - Homework 7

An Empirical Evaluation of Distributed Key/Value Storage Systems

Sagar Shekhargouda Patil -A20501427

Issa Muradli -A20474520

Problem:

The assignment's aim was to compare several distributed key-value storage systems. We selected to compare Cassandra, Redis and MongoDB distributed stores in accordance with the studies conducted on ZHT, Cassandra, and Memcached.

Methodology:

On Chameleon's Compute Haswell nodes, we tested the systems using containers with 3 cores, 12 GB of RAM, and 24 GB of disk space. For all systems, nodes were assessed individually as well as in clusters of 2, 4, and 8. We set up a cluster of config servers, query routers, and shard servers for MongoDB assessment.

A 10000-record workload that had 100 bytes per record (10 bytes for the key and 90 bytes for the value) was generated in the python script itself randomly. The systems' latency and throughput were calculated and averaged throughout the insert, lookup, and delete processes.

ZHT, Cassandra, Redis and MongoDB:

An in-memory data structure called the **ZHT**, or Zero Hop Hash Table, was created utilizing distributed systems. The high availability, scalability, fault tolerance, high throughput, and low latency features were designed for high-end computing systems. In order to promote the transition from petascale to exascale, new storage systems were introduced that do not entirely separate storage and computation nodes. The key-value mappings are distributed across nodes using the SDBM hash function, which minimizes the impact of any changes. ZHT offers four operations: add, lookup, delete, and insert (key, value) . For lock-free concurrent alterations and effective concurrent metadata management, the append operation is crucial. Typically, keys are mutable ASCII strings, and values can be complex objects of various sizes.

Every node in ZHT holds knowledge about every other node in a static membership. Nodes may join and exit in a dynamic ecosystem with little to no disturbance. With the help of its IP address and port, each ZHT instance may be located. A manager service is running on each node and is in charge of starting and stopping the system as well as handling membership tables and partition migration. Nodes can leave on failure, which is recognized by a client on timeout, and is updated by the manager. Nodes can also depart on plan, in which case the admin changes the membership table. Since it is more

effective than rehashing several key-value pairs, partitions are completely transferred. Both TCP and UDP can be used to create ZHT, however connectionless protocols are preferable due to their less expensive setup. Clients communicate with a single replica to ensure consistency, while replication is used to increase dependability.

Apache Cassandra is a highly available, high-performance, distributed NoSQL database designed to handle large amounts of data. Runs on multiple commodity servers with no single point of failure. Clusters can be distributed across regions using masterless replication, resulting in low latency. Each node has the same role. Optionally, you can configure replication for redundancy, failover, and disaster recovery. Failed nodes can be replaced without downtime.The Cassandra Query Language (CQL), a structured query language, may be used for operations (SQL). It is a wide-column store, which combines key-value and tabular DBMS features. Tables may be added, removed, and changed without causing any disruptions during runtime. Each key is associated with a set of columns that represent the values of the objects, known as column families. A Java Management Extensions (JMX)-compliant nodetool software manages and keeps track of the clusters in the Java-based Cassandra system.
It can utilize either a straightforward replication approach for usage in a single data center or a network topology replication technique for cluster deployment across several data centers. A commit log is used to guarantee data persistence during synchronous or asynchronous replication. For all the tables it oversees, each node keeps indexes. As a result, managing these indexes will require greater overhead with huge data volumes.

Redis (Remote Dictionary Server) is a distributed, in-memory key-value database, cache, and message broker with configurable durability. Redis is an in-memory data structure store. Various abstract data structures, including strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices, are supported by Redis.
Redis is credited with popularizing the notion of a system that functions as both a cache and a store. It was created with the intention that data would always be changed and read from the main computer memory while also being saved on disk in a manner that was not ideal for random data access. Only after the system restarts does the formatted data be rebuilt into memory.
In addition, Redis offers a data model that is quite distinct from that of a relational database management system (RDBMS). User commands instead specify particular actions that can be carried out on specified abstract data types rather than a query to be done by the database engine.
Data must thus be kept in a fashion that will allow for quick retrieval in the future. The retrieval is carried out independently of the database system's secondary indexes, aggregations, or other typical characteristics of conventional RDBMS. In order for the parent process to continue serving clients while the child process produces an in-memory copy of the data on disk, the Redis implementation heavily utilizes the fork system call.
A NoSQL document database with excellent scalability and flexibility is **MongoDB**. Indexing is a technique used to boost efficiency. The _id field is used to identify a document, and an index is made for

it. Each document has a unique structure and set of data fields that might change over time. The files are kept in Binary JSON format (BSON). Range queries, regular expressions, and user-defined JS functions are examples of queries that can return document fields or random results samples.
With replica sets that include two or more copies of the data, a master-slave replication technique is used to achieve high availability. Secondary replicas keep copies of the primary data while reads and writes are done on the primary replica. A secondary replica is chosen through an election procedure to serve as the primary replica in the event of the primary replica failing. Sharding is used to scale horizontally. The distribution of data among various shards is controlled by a user-selected shard key. Additionally, MongoDB provides drivers for popular frameworks and programming languages.

There are two distributed NoSQL open-source databases: MongoDB and Cassandra. They offer horizontal partitioning and do not adhere to the conventional ACID features. Cassandra is more of a conventional data model, but MongoDB offers a rich object-oriented approach. In terms of data architectures, MongoDB is more adaptable than Cassandra. In situations containing a lot of data, Cassandra can perform better than MongoDB.

ZHT	**Cassandra**	**MongoDB**	**Redis**
In-Memory Data Structure	Flexible wide-column	Stored as BSON Document Files	open-source, in-memory, data structure store”
Lock-Free Concurrent	No Locks	Multi-Granularity Locking	Distribute Locks
Batch-Oriented, Dynamic	Masterless / Multiple Masters	Master-Slave Replication	Master-Slave Replication
Built-In Replication	Built-In Replication Built-In Replication	Built-In Replication	primary-replica architecture and supports asynchronous replication
Implemented in C/C++	Implemented in Dynamo(Java) Multiple Programming Language Support	Implemented in C++, JavaScript, Python Multiple Programming Language Support	written in ANSI C
	Structured and	Structured and	supports unstructured

	Unstructured Data	Unstructured Data	as well structured data
Partitions, Manager Service, Hashing Functions	Ad-Hoc Queries, Aggregation, Collections, File Storage, Indexing, Load Balancing, Replication, And Transactions	Clusters, Commit Logs, Data Centers, Memory Tables	real-time analytics

Results and Observations:

System/Scale	1	2	4	8
Cassandra	**1.2ms**	**0.74ms**	**0.31ms**	**0.13ms**
MongoDB	**0.83ms**	**21.7ms**	**19.70ms**	**NA**
Redis	**0.69ms**	**0.52ms**	**0.45ms**	**0.40ms**

Performance evaluation of Cassandra, MongoDB and Redis [Latency(ms) vs Scale]

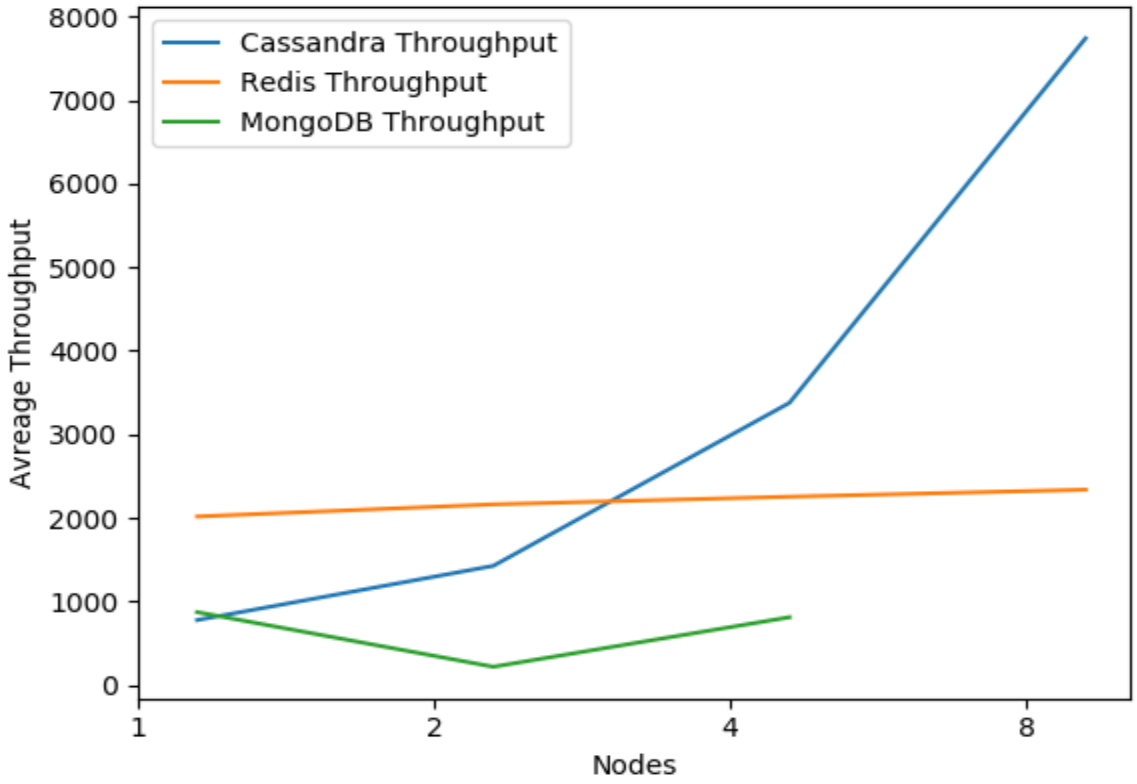
System/Scale	1	2	4	8
Cassandra	**780**	**1433**	**3476**	**7820**
MongoDB	**875**	**234**	**813**	**NA**
Redis	**2108**	**2162**	**2251**	**2347**

Performance evaluation of Cassandra, MongoDB and Redis [Throughput(ops/sec) vs Scale]

Observations:

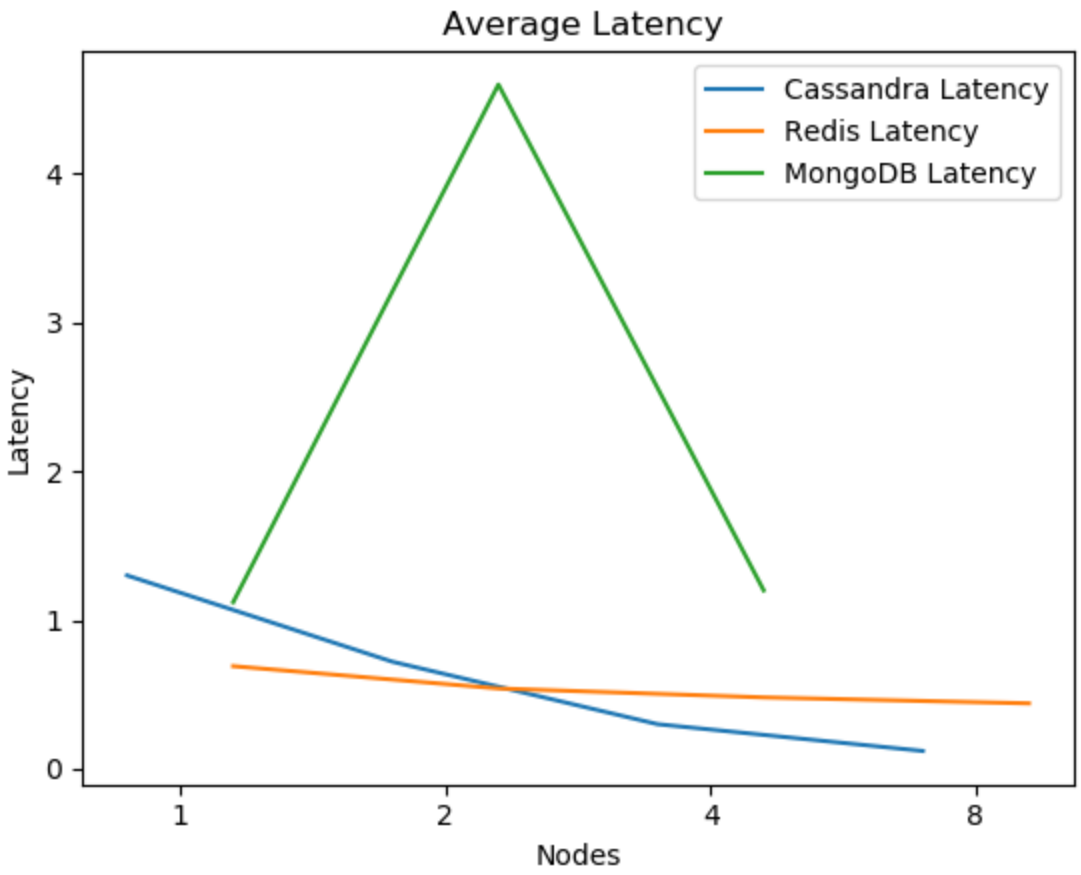
According to our findings, Cassandra exceeds MongoDB in terms of throughput by a wide margin. However, the MongoDB clusters showed lower latency. The latency in MongoDB was trending lower (After Two Nodes). This suggests that when cluster size is increased, MongoDB performance improves. According to the published data, we saw an improvement in throughput as cluster size increased and a negligible increase in latency. Our tests showed that the throughput in MongoDB increased as the cluster size increased. To increase throughput, batch operations might be a smart strategy. Performance may be considerably enhanced by this. Concurrent query execution is a different strategy.

1). Throughput:



From the above graph, Redis has the greatest throughput for single-node setup among the three, as can be seen in the graph above. Cassandra's throughput appears to increase gradually as the number of nodes rises, whereas MongoDB's throughput decreases for two-node clusters before rising steadily. This is because configuring a configuration server and router for a two-node cluster significantly reduces MongoDB's throughput, while Redis performs consistently as the number of nodes rises.

2). Latency



From the graph, Cassandra's latency was the greatest for a setup with a single node when compared to MongoDB and Redis. Additionally, we can observe that MongoDB latency is lowest for four-node clusters and increases dramatically for two-node clusters. Thus, we may infer that MongoDB will have lower latency than the other two DBs for the high number of nodes.