## Compare and contrast your 3 evaluated systems to ZHT

Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database) for managing large amounts of structured data distributed around the world. It provides high availability services with no single point of failure.
Its main advantages are scalability, fault tolerance and consistency. It is a column-oriented database with a distribution design based on Amazon's Dynamo and its data model on Google's Bigtable.
Cassandra implements the Dynamo-style replication model without a single point of failure, but adds a more robust "column family" data model, where the column family is stored in a single file on disk. Therefore, to optimize performance, it is important to keep the columns you may want to query in the same Column Family, and a supercolumn may help here. A supercolumn is a special column, and as such, it is also a key-value pair. But a supercolumn stores a mapping of subcolumns. column is the basic data structure of Cassandra and has three values, the key or column name, the value, and the timestamp.
Cassandra is designed to handle big data workloads on multiple nodes without any single point of failure. cassandra is linearly scalable, meaning that it can increase throughput as the number of nodes in a cluster increases. Cassandra has a peer-to-peer distributed system among its nodes, and data is distributed among all nodes in the cluster. All nodes in the cluster play the same role. Each node is independent and interconnects to other nodes simultaneously. Each node in the cluster can accept read and write requests, regardless of where the data is physically located in the cluster. In the event of a node failure, read/write requests can be handled from other nodes in the network.
The components of the Cassandra component include Node, Data center, Cluster, Commit log, Mem-table, SSTable, Bloom filter, and the Cassandra query language. We can access Cassandra through its nodes using the Cassandra Query Language (CQL). CQL treats the database Keyspace as a table and uses cqlsh to perform queries.
The Cassandra database is distributed across multiple computers. The outermost container is called a cluster. for fault handling, each node contains a replica, which is responsible if a failure occurs. Cassandra arranges nodes in a cluster in a ring format and allocates data to them. keyspace is the outermost container for data in Cassandra. The basic properties of a Keyspace in Cassandra are Replication factor - The number of computers in the cluster that will receive copies of the same data.
Replica placement strategy - The strategy for placing replicas in the ring. We have policies such as simple policy (rack-aware policy), old network topology policy (rack-aware policy) and network topology policy (data center sharing policy).
Column families - A Keyspace is a container for one or more Column families, which in turn are containers for sets of rows. Column families represent the structure of the data. Each Keyspace has at least one Column family, and there are usually many Column families.

https://cassandra.apache.org/doc/latest/cassandra/new/index.html

Redis is an open source (BSD licensed), in-memory data structure store used as a database, cache, message broker, and streaming engine. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. Redis allows to run atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing an element to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set. To achieve top performance, Redis works with an in-memory dataset. Redis can persist the data either by periodically dumping the dataset to disk or by

appending each command to a disk-based log. It is possible to disable persistence in case of a feature-rich, networked, in-memory cache. Redis supports asynchronous replication, with fast non-blocking synchronization and auto-reconnection with partial resynchronization on net split. Redis also includes Transactions, Pub/Sub, Lua scripting, Keys with a limited time-to-live, LRU eviction of keys, Automatic failover. Redis is written in ANSI C and works on most POSIX systems like Linux, *BSD, and Mac OS X, without external dependencies.[1]

MongoDB is a document database designed for ease of application development and scaling. A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents. The advantages of using documents are documents correspond to native data types in many programming languages, embedded documents and arrays reduce need for expensive joins, dynamic schema supports fluent polymorphism. MongoDB provides high performance data persistence. In particular, support for embedded data models reduces I/O activity on database system, indexes support faster queries and can include keys from embedded documents and arrays. MongoDB's replication facility, called replica set, provides automatic failover, data redundancy. A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability. MongoDB provides horizontal scalability as part of its core functionality. MongoDB supports multiple storage engines. MongoDB provides pluggable storage engine API that allows third parties to develop storage engines for MongoDB.[2]

Zero-hop distributed hash table (ZHT) is a tuned hash table implementation. It is modified for the specific requirements of high-end computing (e.g., trustworthy/reliable hardware, fast networks, non-existent “churn”, low latencies, and scientific computing data-access patterns). The purpose of ZHT is to form a fundamental block for future distributed systems, thereby delivering excellent availability, fault tolerance, high throughput, scalability, persistence, and low latencies. The various important features of ZHT make it a better candidate than other distributed hash tables and key-value stores. The feature of ZHT include being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication and by handling failures gracefully and efficiently propagating events throughout the system, a customizable consistent hashing function, supporting persistence for better recoverability in case of faults, scalable, and supporting unconventional operations such as append the data along with the traditional operations of a hash table to insert/lookup/remove.[3]

| Name | Performance (Ranked) | Scalability (Ranked) | Ease of use (Ranked) |
|---|---|---|---|
| Cassandra | 2 | 2 | 2 |
| Redis | 4 | 4 | 4 |
| Mongodb | 3 | 3 | 3 |
| ZHT | 1 | 1 | 1 |

| Name | Implementation | Routing time | Persistance | Dynamic membership | Append |
|---|---|---|---|---|---|
| Cassandra | Java | log(N) | Yes | Yes | No |
| Redis | ANSI C | log(N) | Yes | Yes | No |
| Mongodb | C++ | log(N) | Yes | Yes | No |

[1] Introduction to Redis: https://redis.io/docs/about/
[2] Introduction to Mongodb: https://www.mongodb.com/docs/manual/introduction/
[3] Introduction to ZHT: Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang,
Ioan Raicu. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IEEE
International Parallel & Distributed Processing Symposium (IPDPS) 2013

| | | | | | |
|---|---|---|---|---|---|
| ZHT | C++ | constant | Yes | Yes | Yes |

Similarities and Differences between Cassandra, Redis, Mongodb and ZHT:
While all are NoSQL databases, Redis is an in-memory data store that supports many different data types, used as a database, cache, and message broker. Cassandra is a distributed key-value store. MongoDB is written in C++, while Cassandra is written java, Redis in C, and ZHT is implemented in C++. MongoDB stores data in JSON format while Redis stores in key-value stores, Cassandra in a wide-columns store and ZHT uses FusionFS files system. MongoDB scaling is relatively easier than either Redis or Cassandra, ZHT scales better than the other databases. While ZHT is more flexible than Mongodb, MongoDB also allows more flexibility than Redis and Cassandra. Cassandra uses a wide-column store as a primary database model, making it easy to store huge databases. All of these use key-value storage pair. Cassandra is more focused on giving you stability. Redis is much faster than Cassandra, but it gets slower if you use it for huge data sets and is ideally suited for rapidly changing datasets. Redis is more useful when you have in-memory data storage, vertically scalable, and read-oriented data structure and dynamic database. Redis is more useful when you have in-memory data storage, vertically scalable database. Also, it is useful when the database is read-oriented & dynamic, while ZHT, Cassandra, and Mongodb are not in-memory data.

## Experimental procedure:

**Cassandra:**
We need to download the java sdk first, then download the Cassandra database, and after that we configure the cluster, the Cassandra cluster adding process will be discovered automatically, so we need to ensure that node1 can connect to other cluster nodes via ssh.

Then modify the **cassandra.yaml** file, modify the **listen_address**, **seed_provider** and **rpc_address** configuration to node1's ip address, in addition, we configure the host(**/etc/hosts**) for all node.

**Redis**
We downloaded the redis-plus-plus shell and along with its supporting packages and libraries after that we configured the clusters so that redis nodes can talk with each other.

Then we modified the **redis config** file, set the **listen_address**, **seed_provider** and **rpc_address** configuration to 1st node's IP address. Also, we configured the host(**/etc/hosts**) for all nodes.

**Mongodb**
We downloaded the mongodb shell (mongosh) and along with its supporting packages and libraries after that we configured the clusters so that all nodes can talk with each other.

Then we modified the **mongodb config** file, set the **listen_address**, **seed_provider** and **rpc_address** configuration to 1st node's IP address. Also, we configured the host(**/etc/hosts**) for all nodes.

## Experimental Result:

Latency - insert (ms)

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 0.742 | 1.035 | 1.141 | 1.151 |

| | | | | |
|---|---|---|---|---|
| Redis | 0.69 | 0.983 | 1.089 | 1.099 |
| Mongodb | 0.16 | 0.453 | 0.559 | 0.569 |

Latency - lookup(ms)

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 0.925 | 1.276 | 1.378 | 1.405 |
| Redis | 0.55 | 0.901 | 1.003 | 1.03 |
| Mongodb | 0.12 | 0.471 | 0.573 | 0.6 |

Latency - remove(ms)

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 0.653 | 0.934 | 0.987 | 1.076 |
| Redis | 0.44 | 0.721 | 0.774 | 0.863 |
| Mongodb | 0.19 | 0.471 | 0.524 | 0.613 |

Latency - Overall(ms) - average

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 0.773 | 1.082 | 1.168 | 1.211 |
| Redis | 0.56 | 0.869 | 0.955 | 0.998 |
| Mongodb | 0.157 | 0.466 | 0.552 | 0.595 |

Throughput - insert

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 1347 | 1932 | 3505 | 6950 |
| Redis | 1449 | 2034 | 3672 | 7280 |
| Mongodb | 6250 | 4416 | 7156 | 14056 |

Throughput - lookup

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 1081 | 1567 | 2902 | 5693 |
| Redis | 1818 | 2220 | 3988 | 7768 |
| Mongodb | 8333 | 4246 | 6980 | 13336 |

Throughput- remove

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 1531 | 2141 | 4052 | 7434 |
| Redis | 2272 | 2774 | 5168 | 9272 |
| Mongodb | 5263 | 4246 | 7632 | 13048 |

Throughput- Overall

| **System/Scale** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Cassandra | 1319 | 1880 | 3486 | 6692 |
| Redis | 1785 | 2302 | 4188 | 8016 |
| Mongodb | 6369 | 4292 | 7248 | 13448 |

We expect the result to be faster for redis, but according to the test results, we found that mongodb is very fast for key, value storage, multiple nodes will lead to longer latency, but since the increase of nodes will make Throughput increase, so the more nodes, it will make the data insertion and deletion query time grow, but it is beneficial for large data volume.