CS4224: Distributed Databases

Benchmarking Distributed Databases

HBase and Cassandra

Submitted By:
Team 14
Divya Bakthavatchalu (A0106481L)
Naman Aggarwal (A0120045B)
Shantanu Alshi (A0120011N)

1 OVERVIEW

Data in modern computing applications is associated with high velocity, large volume, high variety also known as 3 V's of Big-data. This exponential increase of data storage, processing and analysis has urged the NoSQL movement away from conventional relational data model. NoSQL databases are increasingly used in real time web applications and also big data applications. In a broad sense, these databases are classified as Column based, Document stores, key-value stores and Graph database based on their data storage techniques and other features.

In addition to the mentioned characteristics of data in modern applications, there is an increased need to ensure high availability and reliability with flexibility. Data demands to be organized in fragments reflecting organizational structure while still providing high performance to common queries. This calls for data to be distributed across interconnected computers located in the same physical location or dispersed over a network of computers.

In this report, we will analyze two of such NoSQL databases - HBase and Cassandra in distributed context. Both Cassandra and HBase are key-value stores in which data is stored in the form of key-values in the filesystem. We will compare the databases in terms of some of the important considerations while choosing a Distributed database such as Query processing, Concurrency Control, Replication, Recovery, and Fault Tolerance. Finally, we will benchmark the two database using the Yahoo Cloud System Benchmark tool (YCSB).

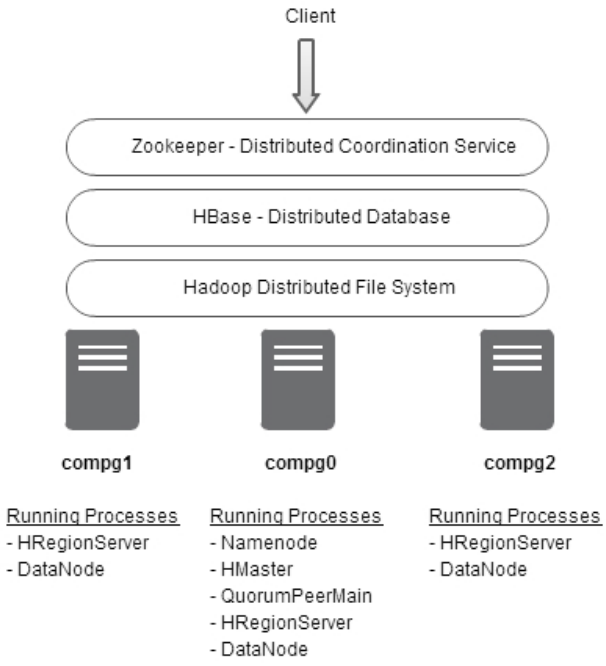
2 TASKS ALLOCATION

S.No	Tasks	Contributed by
1	Planning and Analysis	ALL
2	HBase installation and Setup	Naman Aggarwal
3	Cassandra Installation and setup	Shantanu Alshi
4	Workload run and analysis of experimental results	ALL
5	Documentation	ALL

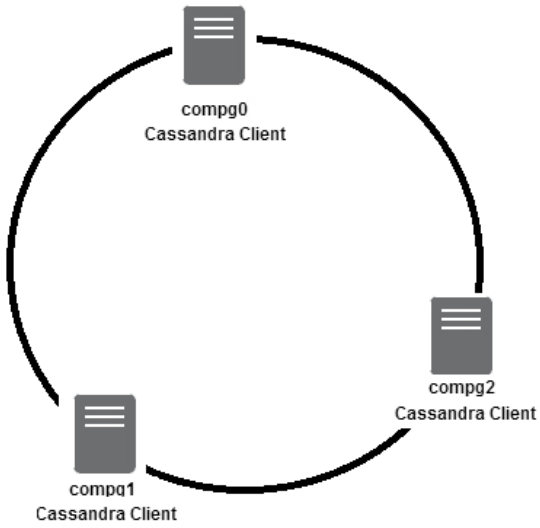
3 PURPOSE

The purpose of this project and report is to study the internals and working of two NoSQL databases and benchmark them using the Yahoo Cloud Servicing Benchmark (YCSB) tool. The two popular NoSQL databases that we will be evaluating as a part of this study are HBase and Cassandra. HBase is a non-relational and open source distributed database. It is written in Java and modelled after Google's BigTable. It is a part of the Apache Foundations Hadoop Project and runs on top of the Hadoop File System(HDFS). On the other hand, Cassandra is also an open source distributed database management system developed by Facebook.

4 GENERAL ARCHITECTURE



HBase Architecture



Cassandra Architecture

4.1 HBASE

HBase is a distributed storage system that runs on a cluster of computers. The cluster can be built using commodity hardware. Each node of the cluster contributes in terms of storage space, cache and computation. HBase is based on the master-slave architecture. HBase slaves called RegionServers are used to store data. Reads and writes are directly performed on the RegionServers. HBase master performs administrative work such as splitting, replication etc.

Each RegionServer stores data on a basic unit called region which is a contiguous range of rows stored together. Data is stored in the form of table. Each row in the table is uniquely identified by a row key and all rows are sorted lexicographically by the key. Columns in a row are grouped together in something called as column families. The column family has to be specified during the creation of table and cannot be changed. HBase puts the limit on number of column families to be in tens, however, most of the use cases do not require more than 10 column families. All the columns in a column family are stored together in the same low-level storage file, called HFile.

4.2 CASSANDRA

Cassandra is a NoSQL database that is perfect for managing large amount of data across multiple data centers and the cloud. Cassandra is massively scalable and delivers continuous availability, linear scalability and operational simplicity. It boasts a powerful data model with no single point of failure. As opposed to HBase, Cassandra is a 'masterless' architecture, meaning that nodes are not distinguished as master/slave, rather all nodes are the same. Replication across this 'ring' guarantees availability of data should a node go down in the cluster. The Cassandra supports Cassandra Query Language (CQL) which is the primary interface to the DBMS. CQL and the Structured Query Language (SQL) share the same abstract idea of tables, except for supporting joins and subqueries), that aids to simplicity.

5 DATA PARTITIONING

5.1 HBASE

HBase uses auto-sharding. Whenever a region becomes too large it is automatically split by the system. Different regions can also be merged in order to reduce the number of storage files. Each region is served by exactly one RegionServer which in turn serves multiple regions. Starting with exactly one region, system monitors to check that it does not exceed the limit. This limit is configurable. Whenever the limit is exceeded the region is divided by the middle key, making roughly two regions of equal sizes.

The master server has the responsibility of assigning regions to the RegionServer and uses Apache Zookeeper – a distributed service to do the task. It also handles the load balancing of regions across the servers to unload the busy servers and move the regions to less occupied ones. The task of splitting the regions is left to the RegionServers themselves.

The split happens very quickly because the system creates two files which act as reference for the new regions (daughter regions), each having half of the data from the original region (parent region). After the process is executed, master is informed about the split so it can move the regions for load balancing reasons.

5.2 CASSANDRA

Data in Cassandra is organized by table and identified by a primary key that is an indicator of the node where data is stored. Replicas are copies of rows and the first write is the first replica.

Cassandra uses Consistent hashing partitions based on the partition key. It uses an order preserving hash function to do this. It places the data on each node according to the value of the partition key and range that the node is responsible for. It also uses 'Virtual Nodes' in order to balance the cluster. This helps Cassandra manage situations when nodes dynamically leave and join the cluster. Within a cluster, virtual nodes are randomly selected and non-contiguous. The placement of any row is determined by the hash of the partition key within many smaller partition ranges that belong to each node.

A partitioner determines how the data is distributed across the nodes in a cluster. Cassandra has the following three partitioners -

- *Murmur3Partitioner* - This is the default configuration and distributes data uniformly across the cluster based on MurmurHash hash values. It provides faster hashing and thus improved performance when compared to the RandomPartitioner. This can be applied to new clusters only. Existing clusters cannot be changed.
- *RandomPartitioner* - This partitioner distributes data uniformly based on the MD5 hash values. Being the default one in previous versions, it is included for backwards compatibility.
- *ByteOrderedPartitioner* - This is included mainly for ordered partitioning. This orders rows lexically by key bytes. Tokens can be directly calculated with the actual values of partition key data using a hexadecimal representation of leading characters in a key. This method is not recommended as it makes load balancing uneven and difficult and can create hotspots due to sequential writes.

We use the default Murmur3 Partitioner in our configuration.

6 QUERY PROCESSING

6.1 HBASE

Regionservers are responsible for all read and write requests for the regions they serve. Clients communicate directly with them to handle the requests.

Client should be able to find the specific RegionServer hosting the row key range on which read/write have to be done. For this purpose HBase provides two special tables called -ROOT- and *.META.*. The -ROOT- table never splits and is served by exactly one RegionServer. The *.META.* table like any other can split across multiple RegionServers. The -ROOT- table points to a region of .META. table that contains the entry for a particular row that needs to be accessed. .META. table contains the entries which a client needs to determine the RegionServer hosting a particular region.Zookeeper keeps the location of the node serving the -ROOT- table. The whole process can be summarized in the following steps -

- Client asks zookeeper for the RegionServer serving -ROOT- table.
- Client then asks -ROOT- table for the RegionServer that has the region for .META. table that can give the location for particular row.
- Client then asks the .META. table for the location for RegionServer that is hosting that particular row.
- Finally client contacts the RegionServer directly reading/writing data on a particular row.

The whole system can be thought of as a 3 level B+ tree. Clients can cache the region location obtained for faster read next time. The client uses recursive discovery in case the cached information is stale.

6.2 CASSANDRA

Cassandra first writes changed data to a commitlog. This commitlog acts as a crash recovery log for data. Write operation is considered successful only after the changed data is written completely to the commitlog so as to guarantee recovery. The write behind cache for Commitlog is periodically synced to the disk every 1000ms which can be changed in the configuration. This enables Cassandra to recover the data on node crash that was lost 1 second within the crash. In addition to that, data is also written to a memory structure called as memtable. Each columnfamily has a separate memtable and retrieves column data from the key. Once this memtable is full, the sorted data is written out sequentially to disk as Sorted String Tables which are immutable in nature.

In order to process a key/column read request, a check is made in the in-memory memtable cache. If not found in the cache, Cassandra reads all the SSTables for that particular column family. It uses bloom filter for each SSTable to determine whether it contains the key. In particular, it depends upon the underlying Operating system for caching SSTable files.

7 CONCURRENCY CONTROL

7.1 HBASE

HBase guarantees row-level atomicity - either the row will be completely written or read or none of it will be written or read. In terms of consistency HBase provides a strong consistency model, that is, client sees the write as soon as it is written.

To provide isolation in the case of concurrent writes, HBase provides a row lock mechanism. The RegionServer provides the row lock, thus ensuring that only the client holding the lock can modify the row. HBase allows to take a lock on row-key whether it exists or not. This ensures atomicity during both create and update. Whenever a put

request is there, server automatically locks the row for the duration of the call. However, instead relying on the implicit server-side locking client can also acquire explicit locks and use them for operations across a row.

A typical write goes through following steps -

Obtain a row lock -> Write to Write Ahead Lock (WAL) -> Update the Memstore -> Release the Row Lock

The locks are held in RegionServer and not in client when a put request is applied. This provides atomicity but not snapshot isolation.

To handle the concurrency control for reads, HBase uses the Multi Version Concurrency Control (MVCC). This ensures that the client can read the changes only after a mutation has been applied to the entire row. When the mutation is in progress, clients will see the previous state of the machine. This is done by giving each write a number and a write only finish after declaring its number. The read just reads the write with the highest number. The steps are as follows-

For Write

- After getting the RowLock, each operation is assigned a write number
- Each data cell in the write stores the write number
- Write operation completes by declaring its write number

For Read

- Each read operation is assigned a timestamp called read point
- The read point is assigned to be the highest number such that all writes with write number <= x has been completed.
- A read then returns the matching data cell whose write number is the largest value less than or equal to read point r.

7.2 CASSANDRA

The traditional RDBMS ACID properties are not followed by Cassandra as these follow eventual consistency. They do not rollback or lock the data as in RDBMS, but they ensure atomicity, isolation and durability with user required level of transaction concurrency.

Atomicity

Cassandra follows row level atomicity (i.e) update or write to each column in a row is considered as a single write operation and it doesn't bundle multiple row updates and doesn't do rollback even if it fails to write on some replica and succeeds on other replicas.

When a data is requested, the most recent update is persisted among all replicas and that data is returned. It is determined by timestamp which is provided by client application. Even multiple clients concurrently updates a single data, the latest timestamp data is returned.

Tunable Consistency

With respect to CAP theorem, Cassandra supports consistency and availability based on the user required level. We can tune to get strong consistency with respect to the CAP theorem and a user can also set the desired level of node response for a DML command or a select query.

Isolation

The latest Cassandra version supports ROW level isolation, i.e other users will not see the other user write to the same row. Thereby, Cassandra gets transactional AID properties where the write are isolated at row level.

Durability

In Cassandra, the writes are durable by recording all the writes both in memory and commit log before success acknowledgement. When the memory tables failed to copy to the disk, the commit logs are replayed and recover the lost writes. The durability is further strengthened because of the data replication on other nodes.

Overall, Cassandra supports atomicity and isolation at the row-level and for the sake of high availability and high performance write, Cassandra trades transactional isolation. It offers ACID transactional property.

8 REPLICATION

8.1 HBASE

Replication provides a disaster recovery solution and can contribute to higher availability. The method HBase uses to replicate data is by sending the writes from one cluster to another whenever they come. This inter-cluster replication is achieved by log-shipping and is done asynchronously. This means that the puts and deletes that go into HLog at the time of write are send to the other cluster for replication. The write on the first cluster is not blocked by the edits that are being replicated. Since, the replication happens asynchronously, It can be done across data centres as latencies of the writes is not affected during replication.

The replication in HBase is done at the column family level. When the column family is configured to be replicated, it sends data to other cluster whenever a write happens to it. The same table name and column family should exist on the other cluster for the replication to happen.

Inter-cluster replication in HBase can be configured to be one of these types -

- Master- slave - In this replication all the writes go to one primary server and then send to the replicated server. Nothing is written directly to the replicated column family in the secondary (slave) server directly. However, HBase does not prevent the writes from happening to the secondary server; this has to be ensured at the application level. If the writes happen to the secondary server they are not propagated to the primary server. The slave cluster can have tables and column families that are not replicated.
- Master-master - In this replication scheme, writes received by either cluster are replicated to one another.
- Cyclic - We can configure more than 2 clusters to replicate among them. The replication between any two clusters can be in the master-slave or master-master configuration.

Replication model is chosen based on the application. For most of the applications, master-slave configuration works just fine. We have chosen the master-slave replication model for our experiment.

8.2 CASSANDRA

Data is replicated to multiple nodes to ensure reliability and fault tolerance. Cassandra provides two replication strategies -

- *SimpleStrategy*: This is used for a single data center only. It places the first replica on a node determined by the partitioner. Any other nodes are placed on the next node in the ring. Factors such as Network topology, rack or data center location are not considered in this strategy.
- *NetworkTopologyStrategy* - This is used when the cluster is deployed across multiple data centers. This method places the replicas in the same data center until encountering the first node in another rack. It strives to put replicas on different racks to ensure more reliability. When deciding the number of replicas in each cluster, it considers the ability of being serving two reads locally without incurring cross data-center latency and failure scenarios. Data center clusters can be either configured as two replicas in each data center or as

three replicas in each data center. Both tolerate the failure of a single node per replication group. The former allows local reads at consistency level of ONE whereas the latter does that with a strong consistency level of LOCAL_QUORUM or even multiple nodes failure with a consistency level of ONE.

We use the SimpleStrategy for our configuration.

9 RECOVERY

9.1 HBASE

HBase prevents data loss in the event of server crash by writing to something called "Write Ahead Log (WAL)" before committing a write. Every server has its own WAL to record the changes as they happen. WAL is nothing but a file on the underlying file system. Most of the time this file system is Hadoop distributed file system which makes HBase as durable as HDFS. A write isn't considered complete until a new entry in the WAL has been created.

If HBase goes down and there was some data that was not flushed to the disk yet, HFile can be recovered by replaying the WAL. This process is automated by the HBase recovery process and user does not have to do anything manually. Each server has its own WAL that is shared across all the tables and column families within that RegionServer.

Whenever a region is opened, either because it was moved from one server to another or it has been just started after a crash, it first checks for the *recovery.edits* directory. If it exists, it start reading the edits it contain. The files are in sorted order by the sequenceID, so that the order is preserved during the recovery. Any edit having sequenceID less than or equal to what has already been flushed to disk is skipped. At the end the memstore is forced to flush the edits to the disk. Finally the files in recovered.edits folder are deleted.

9.2 CASSANDRA

Failure detection in Cassandra is done using the gossip state and history. It uses this to avoid routing client requests to unreachable nodes. Cassandra uses the Phi Accural failure detection algorithm. This protocol is based on the idea that failure detection should be produced by decoupling it from the target application and implementation of heartbeat to detect whether a node is alive or dead. It does not have a fixed threshold for marking nodes as 'failed'. Instead, it uses an accrual detection mechanism to calculate a per-node threshold based on various external factors such as network performance, workload and historical conditions. While gossiping, each node in the cluster maintains a sliding window of inter-arrival times of messages from other nodes in the cluster. Lower values of this threshold increase the probability of an unresponsive node marked as down. As node outages are often transient and rarely denote a permanent departure of that node from the cluster, other nodes periodically try to establish connection with the failed nodes to check if they are up.

10 FAULT TOLERANCE

10.1 HBASE

Whenever a region becomes unavailable due to any of these reasons HBase ensures that the data it was serving is now given by some other RegionServer.

For the purpose of fault tolerance and high availability guarantee, HBase relies on the underlying storage that is Hadoop in most of the cases. HBase stores its data on a single file system. It assumes that all the RegionServers have access to that file system across the cluster. The data written by a single server is visible to all the other RegionServer across the cluster. If a RegionServer goes down the other RegionServer can just read the data from underlying storage system and start serving the data.

HDFS provides HBase with single namespace storage and the datanodes and RegionServer are generally the same in most of the clusters. This means that one can write to underlying storage without much network I/O. HDFS also replicates the data and hence can still serve the data in case of a data node failure.

What happens when the failure is a network partition which causes Hbase master to be separated from the cluster or Zookeeper to be separated from the cluster? If there is a possibility of such a scenario it is better to have backup masters which take over the primary master when it fails, and running zookeeper as an ensemble of nodes (Zookeeper quorum).

10.2 CASSANDRA

A node choses a random token for its position in the ring when it starts for the first time. This mapping is persisted to disk locally and also in any distributed co-ordination service such as Zookeeper if used. This token information is gossiped around all nodes in the clustered as described in Failure detection. Also, as Cassandra is configured such that each row is replicated across multiple Data Centers, allows handling failures at any data center without any outages. Cassandra relies on the advantage of consistent hashing mechanism for fault tolerance. Departure of a node will affect only the immediate nodes and other nodes in the ring remain unaffected.

In case a node is down, once the failure detector marks it, other replicas store the missed writes for a period of time. This works only if hinted handoff is enabled. Also, if a node is down for longer than max_hint_window_in_ms (default 3 hours), the hints are no longer saved. A repair has to be run on recovered nodes that were dead in order to synchronize them with undelivered hints.

11 BENCHMARKING

11.1 SPECIFICATIONS

The entire benchmarking experiment is conducted on Compute nodes provided by School of Computing at National University of Singapore. The specifications are as given in the table below-

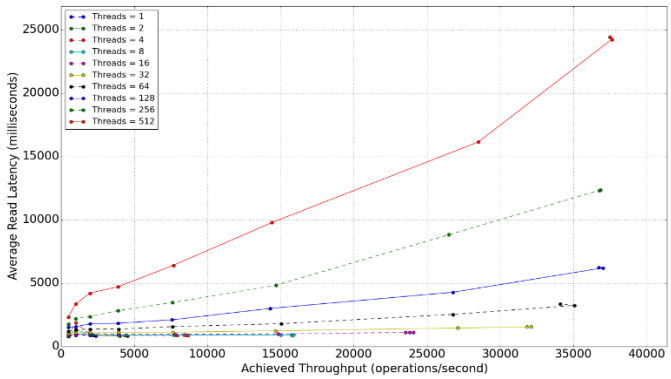
NODES	MEMORY	HARDWARE SPECIFICATIONS	OPERATING SYSTEM
compg0, compg1, compg2	24GB	Supermicro 1RU, 2 x Quad-Core Xeon E5520 2.2GHz	Centos 5.5 64-bit

DATABASE	SOFTWARE VERSION	ADDITIONAL SOFTWARE
HBase	Hbase-0.98.6.1	Hadoop version 2.4.1 Zookeeper Version 3.4.6
Cassandra	Cassandra-1.0.6	----

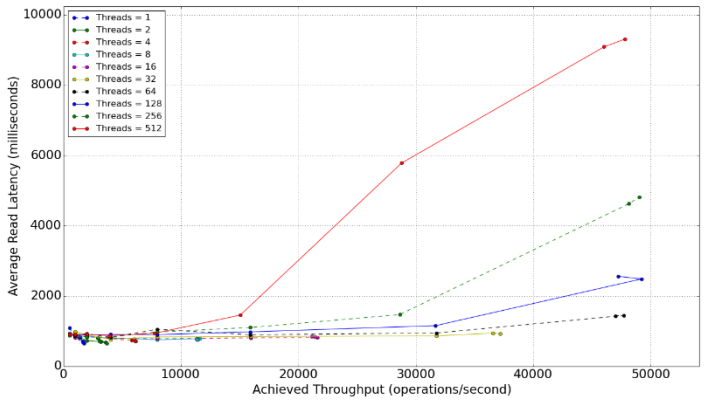
Benchmarking is done using the Yahoo Cloud Servicing Benchmark v0.1.4.

We assume that benchmarking is done on dedicatedly allocated resources and the nodes are not running any compute heavy processes simultaneously at the time of our testing.

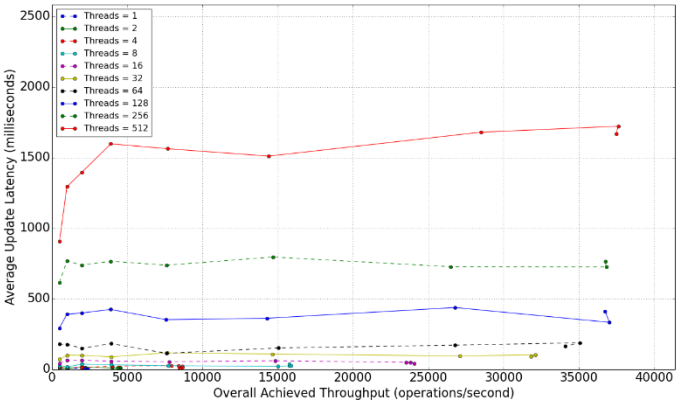
11.2 PERFORMANCE GRAPHS:



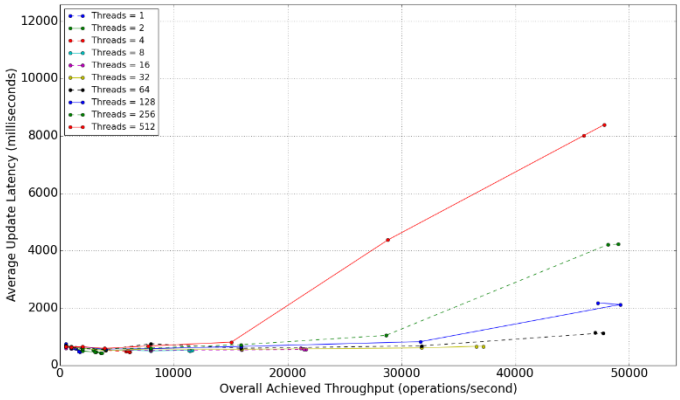
Graph 1. HBase Read Latency vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records



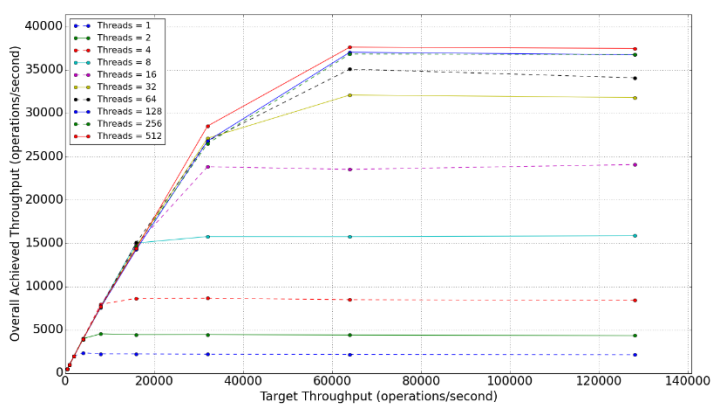
Graph 2. Cassandra Read Latency vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records



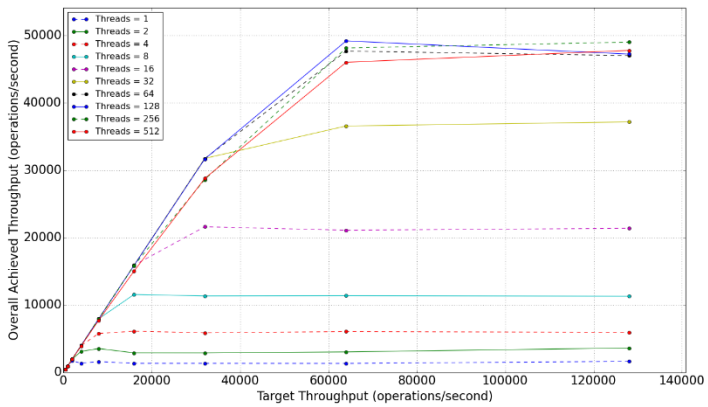
Graph 3. HBase Update Latency vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records



Graph 4. Cassandra Update Latency vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records



Graph 5. HBase Target throughput vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records



Graph 6. Cassandra Target throughput vs vs Achieved Throughput at different number of threads for 300000 operations and 25000000 records

11.3 Discussion of Experiment results

Latency (approx. in ms)	HBase	Cassandra
Minimum update Latency	7	747
Maximum update Latency	1669	8387
Minimum read Latency	507	1099
Maximum read Latency	24418	9307

The following observations can be made on the graph presented in last section-

- Hbase has high average read latency and low average update latency whereas Cassandra has the opposite
- It can be observed that from graphs of both the databases that for lower number of threads the achieved throughput is less no matter how large the target throughput is given.
- Both the update and read latency is small when the throughput is less and increases sharply with throughput for both the databases.
- This increase in latency (both update and read) for Cassandra is relatively sharp than HBase where the increase is gradual.
- For the same number of threads and throughput the read latency in Cassandra is approximately half of the read latency in HBase.
- However, in the case of update latency it is just the opposite. The update latency is approximately twice for same throughput and thread load.
- Graph 5 and 6 clearly show as threads increase, the achieved throughput increases. Also, a particular thread is associated with a threshold after which the throughput remains almost constant. Graph 6 for Cassandra shows that the achieved throughput remains close to 12000 for 8 threads after a threshold throughput of 18000 operations/second.
- The increase too is upper bounded by a threshold after which the increase in achieved throughput remains fairly constant. Graph 6 for Cassandra shows that the increase in achieved throughput decreases after running with 128 threads and stagnate after throughput of 62000 operations/second.

11.4 Problems Faced and Lessons Learnt

- The YCSB for HBase was using old client library. This prevented us from loading the database as YCSB could not move beyond the ZooKeeper. To correct this we added the new client library in the pom file and rebuilt YCSB.
- As YCSB bindings were not compatible with latest version of CQLSH provided with Cassandra 2.x. Therefore, we had to switch to Cassandra 1.0.6 for our project.
- At times, we were not able to execute our tests as processes by other teams were not terminated on time. This affected timelines and also required us to rerun the experiment on certain days.
- As HBase ideally requires name node, data node, backup and HMaster designated on different nodes and that we had only three nodes for our experiment, we had to run the HMaster, Data node and name node simultaneously on one of the nodes. This might have resulted in slight degradation in the performance of that particular node, that otherwise would have not had processes been running on dedicated nodes.
- The access nodes ran Java 1.6 whereas the compg nodes ran Java 1.7. As Cassandra needed Java 1.7, we were not able to run our trial runs on access nodes and had to rely on compg nodes for execution.
- The setup procedure for HBase was quite involved and required a detailed understanding of architecture and components. This resulted in a learning curve and considerable time in database setup. Should anything go wrong in the configuration, we created an automated shell script for this setup for ease of use.
- We also faced some issues in running the experiment as the home directory was at times unmounted for maintenance without prior notice.

- The default record count taken by YCSB is 1000 rows. Initially, we missed specifying the large.dat file during execution in order to override the default record count. This resulted incorrect results and hence missing some of our turns for execution.
- The given requirement of record size was 4KB, however YCSB had no explicit provision for the same. We studied workloada file and understood that fieldcount and fieldlength options that were to be modified to specify this requirement. We set fieldcount 20 as and fieldlength as 204.
- Since we use the shared computing nodes (general task can be run by others) on the experiment day, the default ports that we defined to run the process might be occupied. This required us to manually check and change the ports in the config files.
- Since the compg nodes were shared, the default configured ports for HBase and Cassandra at times had other services running on them. This oftentimes needed changing configuration files.

11.5 NOVEL IDEAS:

- In order to get more insight regarding throughput variation and its effectiveness, we have conducted an experiment at different number of thread count. We have plotted the graph between the target throughput and achieved throughput as shown in the Graphs 5 and 6, it is evident that achieved throughput will be almost constant after a particular threshold as described in Section 11.3 of this report.
- Since HBase and Hadoop (the underlying file system) requires a huge manual effort to set it up, we have completely automated the installation and make it run automatically using shell scripts. The scripts are provided with this report.
- We also automated the generation of graph from log files using python. This script can be used in general for any database. This script is also provided with the report.

12 CONCLUSION AND FUTURE WORK

From the experiments conducted we can conclude that HBase is more suitable when compared with Cassandra for high update intensive applications. Cassandra would be a better option when the numbers of reads are very high. While choosing a database for an application it should be considered that HBase has a single point of failure whereas Cassandra has no single point of failure. Also, while HBase can provide strong consistency at the record level, Cassandra's eventual consistency can be tuned with respect to performance.

As a part of future work, database performance can be studied by benchmarking with respect to number of nodes. This will give insights about the relationship between number of nodes and system attributes. One can also benchmark a particular database with its older versions to see if the newer version provides better performance.

13 REFERENCES

[1] A. Lakshman, P. Malik, *Cassandra – A Decentralized Structured Storage System.* On the web - https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf
[2] Cassandra Datastax Documentation. On the web - http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_about_transactions_c.html
[3] Cassandra Internals documentation. On the web - https://wiki.apache.org/cassandra/ArchitectureInternals
[4] Cassandra Concurrency control. On the web - http://teddyma.gitbooks.io/learncassandra/content/concurrent/concurrency_control.html
[5] L. George, *HBase-The definitive Guide*, O'Reilly
[6] N. Dimiduk, A. Khurana,*HBase in Action*,Manning
[7] Apache HBase Internals: Locking and Multiversion Concurrency Control. On the web - https://blogs.apache.org/hbase/entry/apache_hbase_internals_locking_and