CS4224 – DISTRIBUTED DATABASE PROJECT

HBASE vs. CASSANDRA

Team No - 15

Team Members	Responsibilities
Ashwini Ravi (A0119980W)	• Involved in Complete Set up of HBase and Cassandra • Researched and documented the replication, recovery and fault tolerant architectures of both Hbase and Cassandra • Involved in generating graphs, documenting observations and finally building the conclusion.
Neha Chhabra (A0110309X)	• Involved in Complete Set up of HBase and Cassandra • Researched and documented the Concurrency Control architecture, Query Processing of both Hbase and Cassandra • Involved in generating graphs, documenting observations and finally building the conclusion.
Raghavendhra Balaraman (A0123443R)	• Involved in Complete Set up of HBase and Cassandra. • Researched and Document the Data model and Data Fragmentation architecture of both HBase and Cassandra. • Involved in generating graphs, documenting observations and finally building the conclusion.

1. Introduction:

To support applications that store and access large chunk of data, called as ‘Big Data’, we require a new database management system that supports simpler and faster access to billions of data. As a result, a lot of such distributed database systems are being developed. HBase and Cassandra are two such systems developed by Apache to handle extremely large data sets.

Both HBase and Cassandra are NoSQL distributed databases in storing and accessing of data. Both prevent loss of data due to failure of cluster nodes using replication. Both ensure durability starting from logging all the write operations to a file. The final data is written to a disk which is SSTable in Cassandra whereas HFile in HBase. Both provide the option of executing commands through shell which is largely used for data manipulation.

In spite of these similarities there are notable differences between HBase and Cassandra architectures. Though the client can connect to any nodes in the cluster, Cassandra requires the user to identify the seed nodes within which the communication takes place. While in HBase, one has to identify the master node which coordinates and monitors the region servers (client). In both, row key is the primary index, but, Cassandra provides another indexing mechanism called Secondary index for easy querying of data which is not seen in HBase. Cassandra implements Gossip Protocol to communicate with other nodes in the cluster where as HBase relies on Zookeeper to coordinate the communication between master and client nodes. HBase provides ACID level complaints whereas Cassandra does not completely support ACID semantics.

2. Data Model:

HBase :

“HBase is a sparse, distributed , persistent multidimensional sorted map”. Indexing in HBase takes place in row key, column key and timestamp. The following are some of the terminologies associated with the HBase data model.

Table: The data in HBase are organized into tables.

Rows: The data in a table is stored in rows which are identified by unique row key. The row keys are nothing but a byte array (array of bytes to store collection of data) sorted lexicographically. Thus row key acts a primary key for any table.

Column Families: Each row in a table is grouped/composed of column families. Column families are defined while creating the table schema as they cannot be easily modified.

Column Qualifier*:* Each column family contains millions of column qualifier or columns. They are also treated as byte array. The columns are referenced by *family:qualifier*

Cell*:* It is the place where the data/values are stored in byte arrays. Each cell can be referenced by *rowkey:family:qualifier*

Timestamp: Each cell stores all the old values. In order to version to the values of a cell, timestamp is used. The cell version versions can be configured for each column family , but the default is 3.

Cassandra:

Keyspaces: In Cassandra, Keyspaces stores the data and the column families. Usually, for each application, Cassandra creates one keyspace.

Column Families: Column Families contain the information about the columns which are actually defined based on the application. The different types of column families are static column family, which is defined by Cassandra, and the dynamic column family which are defined by the users or application. The following are some type of the columns in the column families.

- Standard – Contains only one primary key
- Composite – One or more primary keys
- Expiring - Gets deleted after sometime
- Counter – Keeps track/count of the number of occurrence of events.

Data Types: They are defined when the column families are created. The column names and values of Cassandra are stored as Byte arrays similar to HBase. Validator refers to the data type of a column whereas Comparator, refers to the data type of a column name

Indexes: The primary index in Cassandra is the index of the row keys. These indexes are useful in data partitioning and replicating among the nodes in the cluster. Cassandra allows secondary indexing which refers to the indexes on column value. Secondary indexes facilitates easy and efficient querying

3. Data Fragmentation:

HBase:

HBase stores the data in rows of a table. The table is then divided into chunks of rows called "regions" and is distributed to each node in the cluster. The machines which host these regions are called "Region Servers", and are managed by the RegionServer processes. The region contains 'Stores' which in turn has the column families of the table. At any point, a region is present in only one RegionServer. This ensures strong concurrency in HBase. By default, HBase allots only one region while creating a table. As a result, while initial loading, all the data goes into one region and will not utilize the entire cluster. A region, along with .ROOT. and .META. regions forms a B-Tree data structure in searching a row in a table.

Automatic Partitioning:

HBase automatically splits a region into two regions, if the size of the regions exceed a certain limit. Some of the region split policies are as follows.

- ***Constant Size Region Split Policy:*** It splits the region when the size of the data in its store exceeds the value defined by 'hbase.hregion.max.filesize".The default maximum size as specified by HBase is 10GB. This split method was used for HBase versions less than 0.94

- ***Increasing to Upper Bound Region Split Policy:*** The regions in the region servers are split based on the number of regions in a region server. The size of the store is calculated as follows
  Store Size = min(N^2 * hbase.region.memstore.flushsize, hbase.hregion.max.filesize)
  where, N is the no of regions on one region server. As the number of regions on the region server increases, the size of the split also increases.

- ***KeyPrefixRegionSplitPolicy:*** This policy is used when prefixes are set for the keys. The rows are grouped based on the rowkey prefix. This is also called as 'Row Groups'

Pre Partitioning:

Pre Partioning creates many regions of a table by giving appropriate split points at the table creation time. However, poor split points will end up with data skewing. Proper estimation of key distribution helps attain optimum initial loading performance

Working of Region Split

The write requests are accumulated into 'memstore'. The requests are then written to store files called as 'memory flush', when the size of the 'memstore' is no more empty. When the size f the store files increases, the Region files collates these store files to make them larger files. After every flush, if the RegionSplit policy of HBase decides to split the region, then a request is raised. During splitting, the newly created regions creates symbolic links of the files instead of writing their data again into anther new files, instead a symbolic link like files are created called 'Reference files', which points to the parent store file from where it was split. These reference files are just another data files with only half the records.

The splitting decision which is taken locally by the Region Servers, undergoes many events. The RegionServer informs the HMaster before and after the split. It also keeps track of the newly created daughter regions are to identify and rearrange in HDFS. To also provide rollback during the splitting process, Region Server records the snapshots of the state of the database..

Cassandra :

Cassandra maintains its data through a circular or ring architecture of nodes in a cluster. Tokens are assigned to every node configured in the cluster which identifies the position of the node is a ring and the range of data it holds. The column family data are split based on the row key of the keyspace.

The basic idea is that, the ring is scanned in the clockwise direction till a node whose token number greater than the row key of the table is found. This range of data, called as the region is assigned to a node. In specific, each node in the cluster holds the region of data ranging from itself to the next immediate node in the ring. The following are the major partitioners used in Cassandra.

- Random Partitioner
- Byte Ordered Partitioner

Random Partitioner:

It is the default partitioner used in Cassandra. It uses the concept of '**Consistent Hashing'** to partition the data. The advantage of using consistent hashing could be seen during addition or removal of nodes from the cluster, in which only less data is affected. The consistent hashing is performed by doing MD5 on the row key of the keyspace. The 128 bit hash value lies in the range from $0\text{-}2^{127}$. The region within this range is associated to each node in the cluster. So any data whose row key lies within the allotted range of a node, is responsible for storing that data. Thus,

$$\text{Token Range} = 2^{127} / \text{Number of nodes in the cluster}$$

Byte Order Partitioner:

This type of partitioner of Cassandra offers partitioning based on the key bytes. The tokens are calculated based on the row key and hexadecimal representation of the leading character of the row key. Order partitioner allows to scan rows as they are ordered sequentially. The following are some of the reasons why ordered partitioner is not preferred.

Hotspots : When an application tries to write data to a block of rows at a time, they only affect one node in the cluster

Load balance overhead : The partitions are defined by the administrators. They are calculated based on the number of row keys in the data. It creates lot of load overhead during the distribution of data across the cluster after the initial loading of data.

Load Imbalance with multiple column families: If the keyspace contains multiple column families, they might have different row keys. The ordered partitioner which balances properly for one column can result in uneven distribution of load for other column families.

4. Query Processing:

In distributed databases query processing consists of following phases: local processing, reduction and evaluation. A query is decomposed into sub queries which require operations at geographically divided local databases, then determine the sequence and the sites for such executions so as to minimize the cost of communication and cost of processing. To optimize queries, information related to tables, indexes is required. Also, optimization algorithms play an important role to determine the performance of query processing.

HBase:

HBase offers two query options. It can ***scan the database*** through ordered keys or by using ***Hadoop to perform MapReduce*** (Map Reduce is parallel processing of vast amounts of data). MapReduce and related systems like Pig and Hive can be combined with HBase because it uses hadoop HDFS to store its data. Hbase does not support secondary indexes, but a trigger on a "put" can keep a secondary index up-to-date and not put burden on client application.

Cassandra:

Cassandra supports secondary indexes on column families. Unlike HBase which uses HDFS and thus enabling use of MapReduce. Cassandra can't work well with MapReduce. In both HBase and Cassandra, row key is the primary index and data is stored on disk such that column family members are kept in close proximity to one another. Therefore it becomes important to plan the organization of column families. The columns having similar access patterns should be kept in one column family to keep the query performance high.

5. Concurrency Control

Concurrency control is for coordinating concurrent requests to a database by multiple clients. Transactions are performed on databases and normally ACID properties are followed by the concurrent transactions. ACID properties are

Atomicity - Atomicity ensures for each transaction either all or none. The whole transactions fails even when a part of the transaction fails and the state of the database remains same

Consistency – Consistency ensures that after every transaction the database state must move from a valid state to a valid state.

Isolation – Isolation ensures that the state of the database is such that each transaction is run serially. So one transaction doesn't impact another transaction.

Durability – Durability ensures that when a transaction gets committed, then even in the case of any crash or power failures, the transaction still remains committed.

HBase

HBase ensures ***ACID semantics-per-row*** (for performance reasons) for the data that requires concurrency control. Suppose there are two concurrent writes to HBase.

HBase follows the following Write Path:

1. Write to Write Ahead Log(WAL) for disaster recovery.
2. Update MemStore: Memstore is updated with each cell value.
3. Flush memstore to disk as a file.

In absence of concurrency control, ACID properties might not be followed. So, HBase ensures row level locking to ensure consistent database state.

The steps performed in case of row level locking are:

1. Row Lock Obtained
2. WAL written

3. MemStore updated
4. Row Lock Released

Row level locking ensures consistency in case of Write-Write Synchronization but in case of Read-Write synchronization both read and write would be fighting for the row level locking and thus slowing down the process. To solve the issue, ***MVCC (Multiversion Concurrency Control)*** is maintained for the reads to avoid row level locking.

Thus using MVCC the steps for read-write or write-write become:

1. Row lock obtained
2. New write number acquired
3. WAL written
4. MemStore Updated
5. Write Number Finished
6. Row Lock Released

In detail following steps are performed:

For writes:

1. A write number is allotted to each write operation after acquiring the row lock.
2. Each data cell stores its write number.
3. Once the write operation is completed it declares itself as finished, mentioning its write number.

For reads:

1. A read timestamp is allotted to each read operation, this point is called read point (Y).
2. The read point is assigned the highest integer so that all writes having write number <= Y are complete.
3. On a read call for a record, a matching data cell with write number being the largest value but <= to the read point of r is given back.

Cassandra

Cassandra doesn't comply RDBMS ACID properties strictly. It offers atomicity, durability and isolated transactions at row level and maintains eventual consistency thus empowering user to decide how strong the consistency can be. However, Cassandra does not support combining multiple row updates into a single all-or-none operation. In Cassandra, the latest updated values of a column can be obtained using timestamps. Whenever requesting data, the latest timestamp takes over. So, in case of multiple client session updating same column in a row, the most recent update will persist.

Cassandra provides ***tunable consistency*** which means client application decides the level of consistency to be maintained. The consistency levels can be different for writes and reads. For writes, the consistency level denotes the number of replicas it should successfully write before it sends back the acknowledgement to application. **In Cassandra writes, following levels are attainable**:

ANY: A write can be written to any node.

ALL: All writes are written to the memory table and commit logs. These are replicated to all nodes in the cluster.

EACH_QUORUM: All writes are written to the memory table and commit logs. These are replicated only on quorum of replica nodes in data centers where,

$$Quorum = (Replication\ Factor/2) + 1$$

LOCAL_ONE: At least one replica node in current data center must acknowledge the reception of data.

LOCAL_SERIAL: All writes are written to the memory table and commit logs. These are replicated only on quorum of replica nodes of same data centers.

ONE: All writes are written to at least one of the replica node's commit log and memory table.

In all the cases the data is sent to all replicas even those belonging to all data centers. The number of replicas required to respond with an acknowledgement determines the consistency level.

In Cassandra reads, following levels are attainable

ALL: It takes data from all replicas and returns the record with latest timestamp. If even a single replica doesn't respond the read operation will fail.

EACH_QUORUM: In this, data is read from a quorum of replicas in every data centres and the record with most recent timestamp is returned.

LOCAL_QUORUM: When a quorum of replicas in the current data centre has responded, it takes the record with the most recent timestamp.

LOCAL_ONE: It takes the data from closest replica (as determined by snitch) in the local data centre and returns the response .

ONE: Returns the data from the closest replica

TWO: Returns latest data from two closest replica

THREE: Returns latest data from three closest replica

QUORUM: Returns the most recent record after the data has been sent from quorum of replicas regardless of the data centre

Snitch:

Snitches determines the relative host proximity. Snitches gather information about network topology such that requests can be routed effectively and efficiently. It also enables Cassandra to distribute replicas by combining nodes into racks and data centers.

Simple Snitch is the 'rack unaware snitch' and thus it chooses the next available node moving clockwise in the ring and places a copy of row on it. This is the default snitch. There are other types of snitches too like rack inferring snitch, property file snitch (These are rack aware snitches), EC2 snitch (this is used by Cassandra in cloud setup)

Vector Clocks:

Vector clocks are used by distributed system to avoid conflict resolution. They involve communicating the messages among the nodes in the cluster by sending the logical clock (an integer that is passed across the nodes). Each time when a node updates its replica data, its logical clock is increased by one. This updated data along with the updated logical clock is sent to all nodes in the cluster. When a node receives a message from another node, it updates each field in the received message if the value of the vector clock in the received message is greater than the value of its own vector clock. But usage of vector clock leads to the following disadvantages in a distributed system.

- Performance : The performance of the system reduces as it involves lots of operations to update a single field.
- Usage of vector clocks generates multiple versions of the data when conflicts occurs.
- Vectors clocks only identifies the conflicts and does not resolve it.

Cassandra follows 'last-write-win' approach to increase performance and keep the design simple where the objects are updated by communicating only to the updated fields. Though 'last-write-win' approach may lead to inconsistent data, cassandra trades consistency for performance and thus the vector clocks are not used in Cassandra.

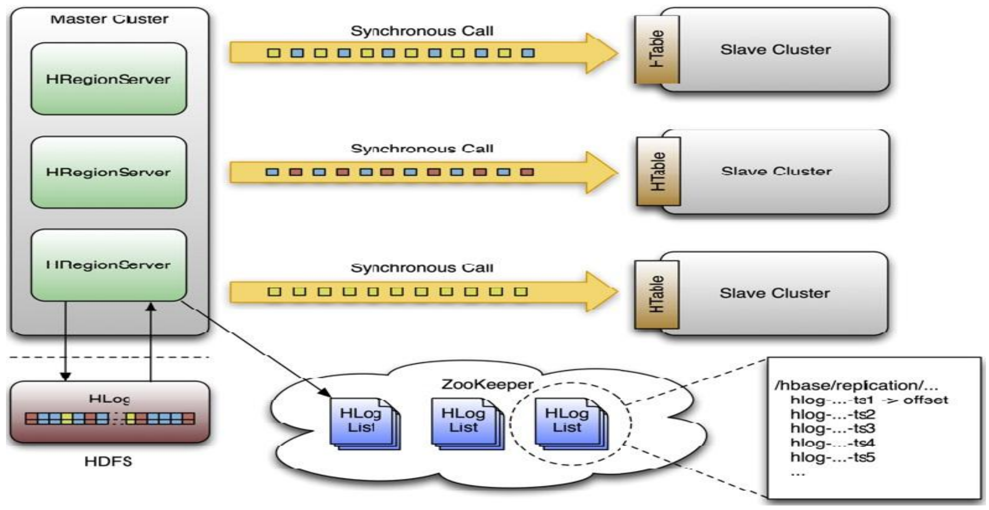
6. Data Replication

HBase

HBase replication defines copying the data from one HBase cluster to another HBase cluster. This replication mechanism is designed for data recovery rather than failover. The cluster getting the data from user applications is called master cluster and the cluster getting the replicated data from the master is called the slave cluster.
In HBase architecture, replication is master push. The most important advantage of this mechanism is that all region server maintains its own Write Ahead Log (WAL). Generally, one master cluster is able to replicate to any number of slave clusters and all region server replicates its own stream of edits.

HBase cluster replication is asynchronous, permitting clusters to be physically far away or to have some distance in availability which means that data between master and slave clusters will not be instantly stable. Rows inserted on the masters are not immediately available or consistent with rows on the slave clusters. The goal is absolute constant.

HBase replication format used in this design is abstractly the same as the statement based replication design used by MySQL. As an alternative of SQL statements, complete Write Ahead Logs consisting of multiple cells inserts receiving from Put and Delete operations on the clients are replicated in order to maintain atomicity.



Write Ahead Logs (WALs) for all region servers should be kept in HDFS as long as they are required to replicate data to any slave cluster. Each Region Server reads from the oldest log it wants to replicate and keeps track of the present position inside ZooKeeper to simplify failure recovery. That position, as well as the queue of Write Ahead Logs to process, may be different for every slave cluster.

The clusters involved in replication can be of different sizes. The master cluster relies on randomization in attempt to balance the stream of replication on the slave clusters. HBase supports master and cyclic replication as well as replication to multiple slaves.

Cassandra:

To ensure fault-tolerance and reliability in distributed environment, Cassandra stores the copies of each row on multiple nodes, called replicas, based on the row key. The number of replicas in the cluster is determined by the "Replication factor" i.e, a replication factor of "n" implies, there are "n" copies of each row stored on different nodes. Here, each replica is treated equally important as there is no master or slave replica. Generally, each write request is rejected when replication factor is greater than the number of nodes in the cluster, whereas the read request is still processed when desired consistency level is met. In order to determine the physical location of each node where the replicas are stored, Cassandra relies on cluster-configured snitch as follows:

- *Simple Replication Strategy:* Used when there is only single data centre cluster. First replica is placed into the node that is determined by the partitioner, followed by the new replicas which are placed into the subsequent node by walking the ring clockwise.
- *Network-Topology Strategy:* Used when the cluster is deployed across multiple-data centres. This strategy is used to determine the number of replicas that need to be placed in each data center based on the real-time applications.

7. Failure Detection and Recovery:

Cassandra handles failure detection and recovery based on the Gossip protocol where each node gossips to the other node in the cluster either directly or indirectly. When each node gossips messages with the other nodes in the cluster, where the gossip messages are maintained in the sliding window based on the inter-arrival times coming from other nodes. Instead of having a fixed threshold value to mark the failure nodes, Cassandra uses the accrual detection mechanism that calculates the pre-node threshold value that includes workload, network performance, historical information etc. When a node is down, it might lose several writes of replication data maintained by that node. When a node is marked as failure by the failure detector, other replicas stores the missed writes for particular amount of time , provided hinted-handoff is enabled. Hints are no longer saved, when a node is down for a longer period of time i.e higher than max_hint_window."Nodetooluitility" or "Opscenter" is used by the administrator, when a node need to be added or removed permanently from Cassandra cluster.

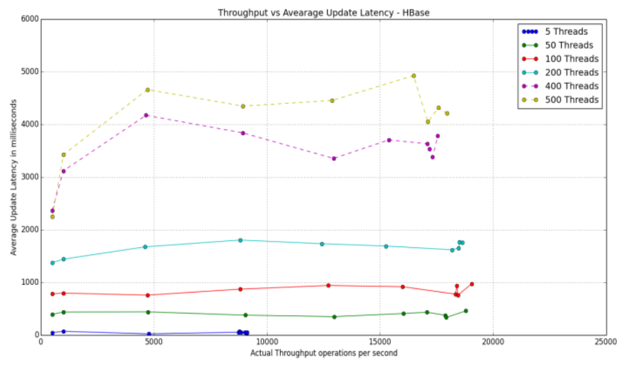
In HBase, there is no global failures i.e when one region server fails, all other regions will be still available. When a node fails, HBase ensures reliability and high availability with minimum downtime by recovering the service quickly without any loss of data. This is achieved using MTTR-Mean Time To Recovery which is defined as " Average time take to automatically recover from the failure", mainly focused on recovery mechanism that is used to identify the failures and helps to restore access as quickly as possible to failed(offline) regions. Using MTTR, node failure detection has been improved, where it lowers the default time-out value.

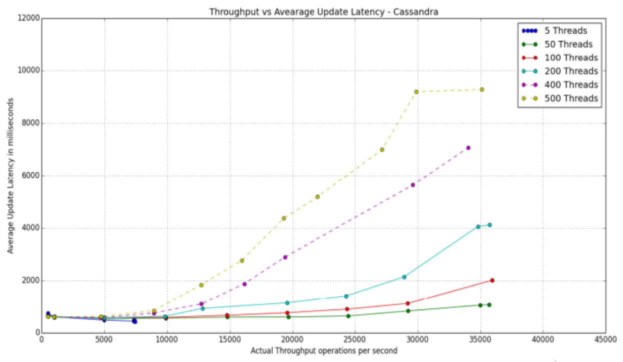
HBase handles failures while ensuring consistency .The data is written in HFiles and saved in HDFS. Later, the blocks of these HFiles is replicated to different nodes across the cluster. The Write-Ahead-Log(WAL) or commit log that is used by HBase is also stored in HDFS which is later replicated to different nodes(3 times by default).

8. Experimental Results:

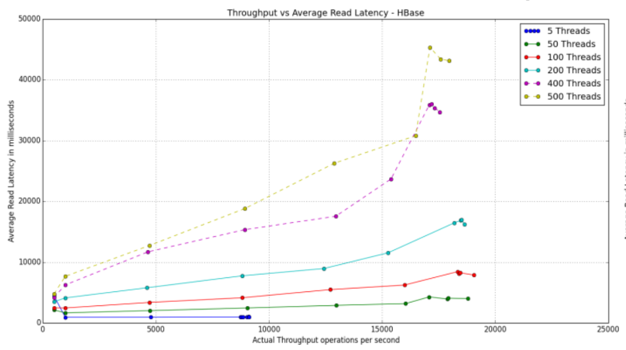
The performance benchmarking of HBase and Cassandra was done using YCSB Workload A for operation count = 100000 and recordcount = 100000000 on a 4 node cluster.

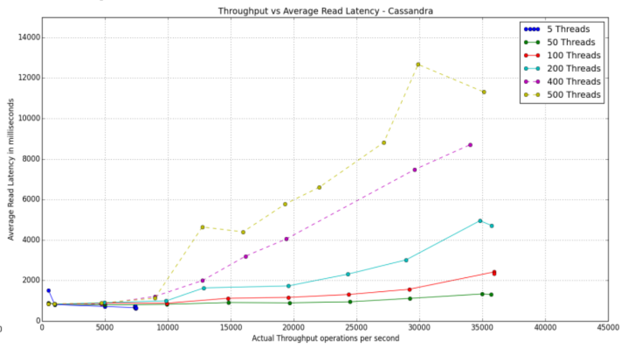
Observed Throughput vs Average Update Latency





Observed Throughput vs Average Read Latency





HBase Observations :

- The update and read latency increases for a throughput value as the number of threads increase.
- The update latency is almost constant with increasing value of throughput for a particular thread value as evident from the plots of threads 5,50,100,200
- The read latency of HBase increases drastically as the throughput increases.
- HBase can achieve a maximum throughput of around 19000 ops/sec (tested up to a target value of 65000)

Cassandra Observations:

- For lower values of throughput the update and read latencies are almost same irrespective of the number of threads
- The update and read latencies increases linearly as the throughput increases with the increase in the number of threads.
- Casssandra can achieve a maximum throughput of around 36000 ops/sec(tested up to a target value of 65000)

From the above observations we derive, at a particular throughput say 17000 for any thread value, the read and update latency of Cassandra is smaller than HBase. The above observations conclude that Cassandra can scale well with a lower read/update latency compared to HBase on a 4 node cluster. The throughput of Cassandra is almost two times as high as the throughput of HBase. The reason could be that Cassandra does not follow master-slave architecture and executes the same code on every seed nodes. This results in no single point of failure thereby achieving higher throughput.

9. Difficulties Faced and Lessons Learnt:

HBase:

- HBase set up did not start as expected and was showing errors in the log. We later identified that the reason was because of missing Zookeeper configuration. We then got to learn more about the use of ZooKeeper in HBase and configured it accordingly.
- HBase showed 'port already in use' errors. We identified the root cause of this problem to be the use of ports in HBase which was already used by some other processes in the system. We later made HBase listen to a different port which solved the problem.
- Hadoop did not start in at least one of the three slave nodes. It was because of different Cluster IDs in different data nodes. The same Cluster ID was then copied to all other data nodes which resolved this problem.
- HBase Master (HMaster) started and lasted only for a small duration. Later we resolved this problem by configuring the 'hbase.rootdir' property with appropriate port number.
- YCSB did not support HBase2.x as it threw errors while building the package. This issue was discussed in the IVLE forum. We followed the suggestions posted there to get this issue resolved.

Cassandra:

- Cassandra while testing in access nodes showed 'java missing errors'. The reason for the error was, Cassandra 2.0 expects java 1.7 whereas access nodes had java 1.6 installed.
- After installing Cassandra2.0, we faced binding problems as YCSB did not support Cassandra 2.0. The solution was suggested in the IVLE forum as per which we migrated to Cassandra1.0.6.
- We were not able to create column families in Cassandra. Later we identified that the reason was because of missing data type. Cassandra expects data type of the column family to be mentioned while creating column families.

10. Conclusion:

Based on the data available and performance report of both the databases we find that both Cassandra and HBase have different use cases. Cassandra is easy to set up, easy to scale incrementally, provides almost optimal reads and writes and has no single point of failure. It will perform best if one is looking for simple set up, maintenance, if the volume of reads and writes is high. Whereas, whenever strict ACID properties are required in transactions and multiple secondary indexes are required it would not perform best. As of now twitter uses Cassandra. On the other hand, HBase is best suited where strict consistency is required, where applications with fast read and writes with easy scalability is required. Facebook uses HBase to maintain photos, chats, messages etc.,

References :

1) Apache Hadoop project: http://hadoop.apache.org.
2) Apache HBase project: http://hbase.apache.org.
3) HBase client API: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/package-summary.html.
4) http://10kloc.wordpress.com/2012/12/27/cassandra-chapter-4-data-partitioning/
5) http://www.geroba.com/cassandra/apache-cassandra-byteorderedpartitioner/
6) http://www.datastax.com/docs/1.0/cluster_architecture/partitioning
7) http://sergeyenin.com/apache-cassandra-1-1-part-1-basic-architecture
8) http://paper.ijcsns.org/07_book/200909/20090918.pdf
9) http://hbase.apache.org/book/architecture.html
10) http://hbase.apache.org/book/regionserver.arch.html
11) http://docs.basho.com/riak/1.2.0/references/appendices/comparisons/Riak-Compared-to-HBase/
12) http://hbase.apache.org/book/secondary.indexes.html
13) http://hadoop-hbase.blogspot.sg/2012/03/acid-in-hbase.html
14) http://teddyma.gitbooks.io/learncassandra/content/replication/turnable_consistency.html
15) https://www.youtube.com/watch?v=ImJn4nc3IPQ
16) https://blogs.apache.org/hbase/entry/apache_hbase_internals_locking_and
17) http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/KeyValue.html.
18) http://lsd.ls.fi.upm.es/lsd/nuevas-tendencias-en-sistemas-distribuidos/HBase_2.pdf
19) http://www.facebook.com/l.php?u=http%3A%2F%2Fdatastax.com%2Fdocumentation%2Fcassandra%2F2.0%2Fcassandra%2Farchitecture%2FarchitectureDataDistributeFailDetect_c.html&h=PAQEcq861
20) Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '06), USENIX, 2006, pp. 205–218