1-1-2014

Performance Analysis Of Scalable Sql And Nosql Databases : A Quantitative Approach

Harish Balasubramanian
Wayne State University,

PERFORMANCE ANALYSIS OF SCALABLE SQL AND NOSQL DATABASES: A QUANTITATIVE APPROACH

by

HARISH BALASUBRAMANIAN

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2014

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor Date

Acknowledgements

I heart fully thank my advisor Dr.Weisong Shi for guiding and supporting me all through my course of master studies and especially during my thesis work by giving invaluable suggestions and providing required resources. I am thankful to Dr.Daniel Grosu and Dr.Zaki Malik for agreeing and adjusting their schedules to be my committee members. I thank my former and current lab mates Tung Nguyen, Guoxing Zhan, Dajun Lu, Quan Zhang, Bing Luo, Sudharshan Raghavan and Shinan Wang for all their support. I thank Andrew Murrell and the IT team who provided support whenever needed. Finally, I thank my parents and my wife for supporting me during the course of masters.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The exponential data growth and increasing number of web users in recent years has pushed the community to create a new kind of data store called "NoSQL" database [2, 5, 6 & 7]. As a result, there are many NoSQL databases available today and choosing one among the set of database is a challenging task for the user. A user can conceptually compare the databases [5, 6] and see if it addresses the user needs, however finding out if a database fulfills the performance requirement needs benchmarking different databases, which is a time consuming task [2]. There are lots of benchmarking reports available in internet and in research papers [9, 29, 30]. Most of the benchmarking reports measure the overall database performance only by throughput and latency. This is an adequate performance analysis but need not to be the end. In this thesis we define some of the new perspectives which also need to be considered during NoSQL performance analysis.

The architecture of NoSQL database is different from traditional database. NoSQL databases works using more than one machine. The performance of a NoSQL database is sum of the performance of individual nodes in the database cluster [5]. **Understanding how a NoSQL database makes use of the individual nodes is important for performance tuning and resource planning.** The performance analysis of the individual nodes can also be used to find the performance bottleneck in the cluster and rectify it. Another perspective is how the NoSQL database balances the differences in performance of the individual nodes. All the nodes in a database cluster may not yield same amount of performance because of the capacity of its resources like disk, network bandwidth and main memory. The nodes in a cluster are not expected to be homogenous. Because in the course of time the potential of storage servers keeps

increasing, so the chance of heterogeneous nodes existing in a database cluster is high. So **how does the NoSQL database manage heterogeneous nodes is another important question**. Because bottleneck nodes may prevent leveraging other nodes in the database cluster and reduce the throughput of the system. The most and genuine utilization of the all nodes in the database cluster could reduce the number of nodes needed.

Both perspectives explained above are practically experienced in our test labs. Noticing that tuning the configuration of the database does not yield the performance expected, led us to look into other reasons and motivated towards this research. In this thesis, three NoSQL databases – HBase, MongoDB, and sharded MySQL were chosen for performance analysis. These three databases differ with each other in its data model, HBase is column oriented storage, MongoDB is document based storage and sharded MySQL is a sharded RDBMS. Using databases which has different architectures for this research gives us an insight how these systems make use of the individual nodes and balance bottleneck nodes. The performance of the database is presented with throughput and latency measured for a period of time and also with individual nodes performance measured using the metric cpu IO wait percentage captured using Ganglia. In addition to that, to extrapolate the degradation in performance with bottleneck nodes, one more individual node was intentionally loaded with lot of disk operations while the benchmarking was running in parallel and the results were captured. The NoSQL benchmarking tool YCSB [2] was used for benchmarking. Different workloads like read-heavy, write-heavy, scan workloads were used for benchmarking. The nodes performance was measured using the open source tool Ganglia [19, 27].

1.1 Contribution

- *We provide some new perspectives to be considered while benchmarking NoSQL database and demonstrate a way to benchmark NoSQL database in a quantitative approach using the existing tools like YCSB, Ganglia.*
- *Benchmark results of three different NoSQL databases are provided and discussed the observations and implications.*
- *We show that how the performance of NoSQL database is affected in a heterogeneous cluster.*

This thesis will be useful to the people who are setting up a new NoSQL database cluster and the people who are troubleshooting their cluster to improve the performance. The approach demonstrated here helps to find out bottleneck nodes in the database cluster and also can interpret which resource of the node is limiting the performance. The results shared in this thesis can also provide some insights for the architects designing the future version of NoSQL databases.

1.2 Terminologies

For better understanding, the terminologies used in this thesis are explained here along with equivalent terms used in the different NoSQL databases used here.

NoSQL Database: General term used in the community to refer scalable cloud data stores. There is no clear definition for the term "NoSQL" but can be roughly understood as "Not Only SQL"[6].

Data node: The nodes configured to store the data/records is referred here as data node. In the NoSQL database cluster, data requests are directed to this node either directly by client

application or through a router component provided by the database. This should not be confused with the term ‘data node’ used in HDFS. In HBase such nodes are called ‘regionservers’, ‘mongod’ in MongoDB. In this thesis the term *data nodes* and *individual nodes* are used with the same meaning. In the result section we use the host name of data nodes to refer to them while discussing.

Shard: A relatively small partition of a large record set. Shards are called regions in HBase. Chapter 2.2.2 discusses more details about sharding.

Cpu IO wait: It's a system metric used to denote the percentage of cpu time spent on waiting for IO operation to complete.

The following chapters were organized as this; Chapter2 discusses the background of NoSQL databases benchmarked in this report and the tools used. Chapter 3 discusses the environmental details, testing strategy and the performance tunings done for the environment and databases. Chapters 4 provide the performance results and discuss about it. Chapter 5 provides the previous work related to this thesis. Chapter 6 discusses about the observation and implications. Chapter 7 provides the conclusion and future work.

Chapter 2 Background

2.1 What is the need for NoSQL Database?

In the recent times data is growing exponentially [5, 6]. Social networking websites, email providers, video hosting websites and many different research organizations are the sources of data generation. Maintaining this extraordinary amount of data has been a big challenge to the companies. Many companies during their startup period used the then prevalent "Relational database" [5]. But as day goes by and the company grows the challenge mentioned above became hard to be handled by relational database [5]. The traditional databases are unable to provide two important features that the industry badly needed,

2.1.1 Scalability

Scalability in database is the ability to handle the data in *terabytes* and *petabytes* scale which may continue to increase in the future and at the same time servicing large number of requests. The current scalability needs has pushed the DBMS to use more than one machine which is called Horizontal scalability. For example, Facebook uses MySQL server which is sharded and running on top of more than four thousand servers [5].

2.1.2 Elasticity

Elasticity is the ability to extend the bandwidth of the running database system. This means the database should be able to adopt new nodes and distribute the data and requests over it without any downtime and with minimal impact in the performance. This feature is very useful for commercial websites where the number of page hits peaks during particular season.

More or less all the traditional database works based on a design of *"One size fits all"* [7]. Those databases were designed compatible for 20 or 30 years old commercial approach [7]. Such databases can only use the resources within single machine such as main memory, disks, and processors. Because of this limitation the traditional databases were not able to support scalability and elasticity.

To overcome this, Google created a new type of database called *"Bigtable"* [1]. It is column oriented scalable database system. It is reportedly running over thousands of node. Such approach of using more than one node for storing data is called "distributed database", in the community it is generally called as NoSQL database. Google's Bigtable is first of its kind and it is proprietary software of Google. There are many type of NoSQL databases in use today.

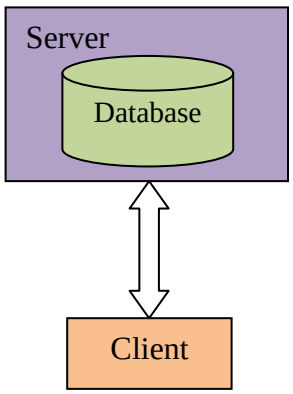

Figure 2.1 Traditional Database

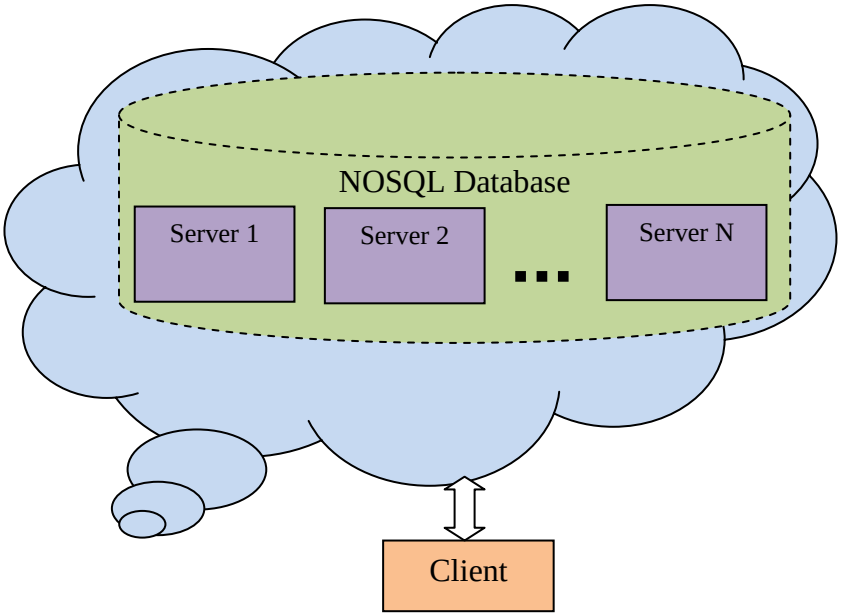

Figure 2.2 NoSQL Database

2.2 Architecture of NoSQL Database

The architecture of NoSQL database uses a cluster of servers. Most of the servers in the cluster play the role of data nodes, the node which maintains data sets. And there are few nodes in the cluster which plays role of monitoring and balancing the cluster, these nodes are called in different names in different databases. In HBase these nodes are called zookeepers, in MongoDB those are called config servers. And there will be metadata node which plays the role of master node assigning data partition/shards to data nodes or acts as a router to the requests.

2.2.1 Types of NoSQL datbases

The NoSQL databases are designed based on the needs of each company. For example, *'Dynamo'* was designed to be highly available storage system for Amazon's online shopping website [15]. *'Bigtable'* was designed to service various applications in Google ranging from real time application to batch processing [1]. Generally NoSQL databases can be classified by the way it stores the data.

- Column oriented database
- Key Value database
- Document database
- And many more

In order to store the data in multiple machines the records have to be partitioned. NoSQL databases commonly use the concept called "Sharding" to split the table records and distribute over multiple machines.

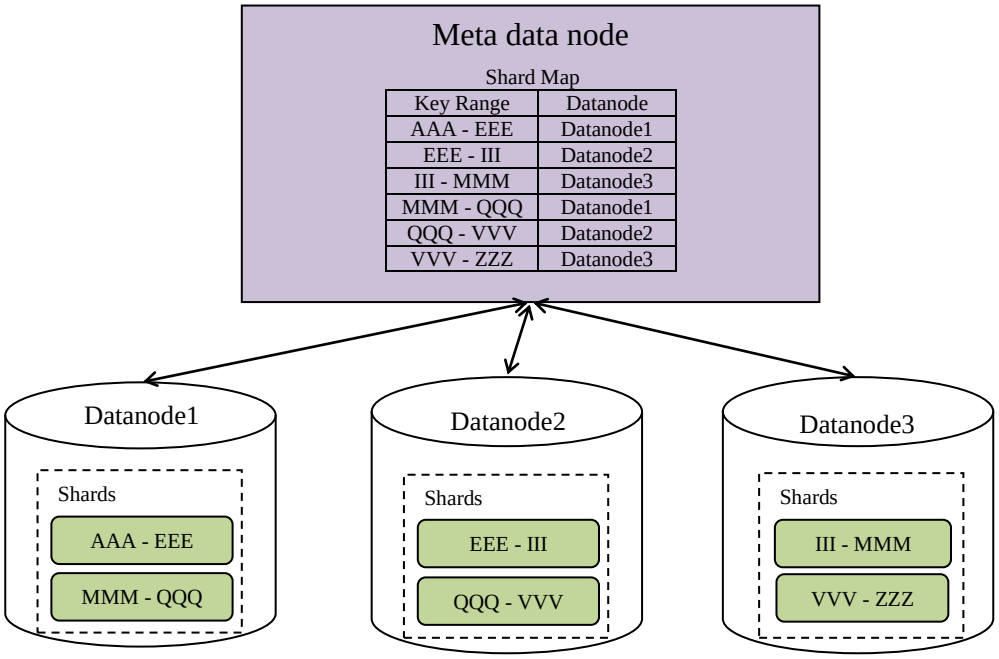

Figure 2.3 An example of sharding architecture

2.2.2 Sharding

Sharding is a concept of splitting a huge record set into multiple relatively small record sets. The record sets are generally split using a shard key which is one of the column in the table. A specific range of key is defined as a shard and any key falls within that range is assigned to that shard. For example, let's say we have customer table and the field "Name" is the shard key, then the records can be split using the range of first character in the name starting from letter 'A' to 'F' and 'G' to 'L' and so forth. A shard can be assigned to one node or set of nodes in case of replication. Generally, a metadata node maintain a map of shard's key range and the node(s) that shard exist. So based on the key of the record which is to be modified or inserted the request is routed to corresponding data node. Sharding can be leveraged by choosing a random and non-repetitive shard key because it gives better distribution. Instead of distributing record by hashing

algorithm, where consecutive rows could be on different machines, having a set of consecutive records in the same machine can help increase sequential read, write and scanning. At the same time random read and write also could achieve high throughput by leveraging multiple machines. Most of the NoSQL databases supports auto sharding which takes splitting and distributing over data nodes.

2.2.3 Trade offs

Even though many companies have started using different type NoSQL databases, there are many complexities in adopting this approach. As already mentioned Facebook [5], tumblr. [10] and a few other companies still manage to scale relational databases by sharding. Though there are some captivating features available in NoSQL databases, it comes with some tradeoffs.

- ACID Properties – NoSQL Databases trades off ACID properties to achieve faster service. For example, HBase writes the updates in the main memory and returns and ensures durability by replicating it into multiple region servers. Also it provides only row level atomicity which would improve the throughput of the database.

- Eventual consistency – According to *Eric Brewer's CAP theorem* a system can only have two of three qualities consistency, availability, and partition tolerance [6]. Some NoSQL databases give up consistency for other two and provide eventual consistency which means updating the replicated copies asynchronously. In such cases there could inconsistent reads. For this reason, some database returns multiple conflicting versions [6] of entity which the client may need to resolve.

- Structured Query language – NoSQL databases does not offer easy to use query language. Usually they provide API to interact with the database. Though some databases some basic level language to query the data through shell or API like MongoDB and HBase.

- Joins – NoSQL does not support joins, even if it does it only provides very basic level of joining. And implementing joins over the NoSQL database is complex and better not to be done [5].

2.3 HBase

HBase [23] is an open source NoSQL database implemented by Apache based on Google's Bigtable design. It is part of Hadoop [24] suite and operates over the distributed file system HDFS [14]. HBase is a multi-dimensional column oriented database i.e. the total number of column in a table is split into one or multiple subsets based on its properties. This subset is called a column family. For example, a student table could have a column family called '*Grades'* which groups all the grades the student earned and another column family called '*Payments'* which groups all payments for different terms. By combining related column as column family and storing them all together makes it easy to store and retrieve columns. HBase also allows multiple versions of data to be stored in the same cell.

HBase partitions the records using auto sharding. Region is a unit record set which comprise all the records which fall within its defined key range. HBase cluster could have multiple Region servers each one of them hosting a set of regions. A Master node is responsible for managing the cluster and maintaining the map of region and region server association. HBase is maintained by Zookeeper servers which is also responsible for proving the routing map to the HBase clients.

2.4 MongoDB

MongoDB is an open source document based NoSQL database developed by 10 gen. MongoDB uses BSON a binary representation of JSON [13] to store the data as documents. Each row is considered as a document. It supports auto sharding, indexing, Map-Reduce, and many other features. Like *Bigtable*, MongoDB also partitions the record using the key range. Each key range is called a "shard" and each shard contains number of chunks. Each shard is maintained by a server called *mongod* or a replication set which is set of mongod servers. The *mongos* server routes for all the requests from client to right mongod server. It is suggested to have 3 configuration server or mongos running in the production environment.[12]. MongoDB supports simple database operations like create, read, update and delete, it also supports some of the basic joining. MongoDB can be connected using any one of the rich set drivers provided by it. MongoDB provides copious tools to monitor the performance of the database system and to improve it. MongoDB does not support ACID transaction but promises atomicity for operation within a single document. MongoDB support journaling which is a mechanism to expedite write operation by copying the operation into journal file and applying write operations as batch.

2.5 MySQL

MySQL is a popular RDBMS owned by Oracle Corporation. Its community version is an open source and can run on wide variety of operating systems. MySQL stores all the data in a single machine and uses B-Tree for indexing. InnoDB is an efficient storage engine and designed to provide higher performance with large amount of database. InnoDB's approach of organizing data on the disk is efficient for serving the queries which filters using primary key. So choosing a primary key which is used in most queries will help achieving higher performance. MySQL provides most of the features that comes with a RDBMS database.

MySQL does not support auto sharding so in order to use it as a distributed database, it was installed in more than one machine and sharding logic was implemented in the benchmarking application. The key was hashed using Java Hashing API and modulo divided by the number of data nodes to find the specific data node.. In contrast to the tablet approach explained in "Sharding" section, this approach places every consecutive record in different machine which costs sequential reads or writes extremely high latency.

2.6 YCSB

Yahoo! Cloud serving benchmark [2] is an open source benchmarking tool implemented in java. The tool was open sourced by Yahoo! which they developed for benchmarking NoSQL database. YCSB benchmarks the database using simple operations like insert, read, write, scan and delete. It provides a set of predifined workloads which can be modified as per the user needs like the percentage of read/write operation and type of random distribution to use while querying data. And also it provides parameters to configure number of threads to use, duration of the test and record count. Apart from this, the source code is flexible to add new workloads as well. YCSB ouput can be captured in a log which would print out the commands used, number of operation and average latency for every 10 seconds and at the end of the test, it gives the overall run time, overall average throughput and latency in different percentile. There are also parameters to capture the time series reports for every configured interval. YCSB can be used to load the database before running benchmark and running parallel workloads.

YCSB supports lot of NoSQL databases like HBase, Cassandra, MongoDB and many more, for which the client module is already implemented and available. Also it is easy to add client implementation for new databases just by implementing few abstract classes. YCSB uses multiple threads while benchmarking the database most of them are worker threads, which is

number of the threads configured by the user and few threads to control and collect statistics. The latency calculated by the YCSB does not include the time spent on collecting the statistics. There is a core component which decides what operation and key to use based on the user configuration and worker threads calls database module with those parameters to perform the operation.

2.7 Ganglia

Ganglia[14] is an open source distributed monitoring tool developed by University of California, Berkeley. It captures the system resource metrics of main memory, processor, disk and network communication. The *gmond* which is client side daemon of ganglia collects the performance metrics from the client system for every fixed interval and reports it to *gmetad* a meta server which stores the metrics in a round robin database. Ganglia provides a web interface to access this information in summary view as well individual nodes for different period of time like hour, day, month and year.

Metric data maintained in the round robin database can be made as graphs using rrdTool[15] All the individual nodes' graphs given in this report was generated using the rrdTool. There are open source modules available to enable the reports for resources like TCP, GPU or specific software like Apache web server, MySQL and as well as for NoSQL databases like Redis, CouchDB, MongoDB. HDFS and HBase can be configured to directly report its statistic to Ganglia.

Chapter 3 Experiment Setup

3.1 Machine Configuration

The cluster had six servers. One of the server was dedicated to run YCSB and in some case to run some of the server component of the database. Each node in the cluster are not of the same capacity. All the nodes are kept in the same server rack and connected to a single network switch.

Node\Metrics	Processors	RAM	Disk	RAID level
hydra1	16 CPUs 2.34 GHZ	16 GB	900 GB	Raid 5
hydra2	16 CPUs 2.34 GHZ	16 GB	900 GB	Raid 5
hydra3	16 CPUs 2.34 GHZ	23 GB	900 GB	Raid 5
hydra4	16 CPUs 2.34 GHZ	22 GB	900 GB	Raid 5
hydra8	8 CPUs 3.81GHZ	8 GB	2 TB	No Raid
hydra9	8 CPUs 3.32 GHZ	8 GB	1 TB	No Raid

Table 3.1: Machine configuration

- *Network Bandwidth* : 1 GBPS

3.2 HBase Configuration

HBase Version 0.94.5 and Hadoop – 1.0.4 was used for benchmarking. About 80% of RAM is allocated for HBase which includes 1 GB heap for Hadoop. Server hydra1 was configured as Hadoop name node and hmaster. All other nodes were configured as data nodes. In

addition, hydra3 and hydra4 were made zookeepers. These two machines have the largest RAM in the cluster.

Most of the performance tuning listed below was followed based on the performance tuning tips [32] section in HBase website.

i. The "hbase.regionserver.handler.count" parameter was set to 20. This parameter is the number of threads that the region server uses to serve the requests from clients. This parameter was tuned based on the performance tuning suggestion from HBase website.

ii. The regions for the table used for benchmarking was pre created. If the regions are not created before loading the data HBase tends to create the regions one by one as data grows and it takes considerable time splitting the data between regions. Also the regions at the end of data loading are huge which may reduce the performance. So 100 regions were created with even distribution of key before loading the data in 5 data nodes each having 20 regions in it.

iii. HBase was developed in Java and allocates and de-allocates heap memory more frequently so garbage collection which is triggered at times could bring down the performance of the database for some moment. To avoid this, concurrent garbage collection was configured as given by the HBase performance tuning tip.

iv. The *bloom filter* feature in HBase was enabled to avoid looking for records in wrong store filter. The filter was enabled for the entire row as the benchmark reads the whole record from the database.

v. The client side buffering was not enabled. Though it may increase the update performance there is also risk of loss of data if the client application goes down before writing the data in the database. However, disabling client side buffering does not show a high impact in write performance of HBase.

3.3 MySQL Configuration

MySQL 5.6.10 was used for this benchmarking. Except hydra1 all other nodes were installed with MySQL server. Since there is no cluster server component in the Shared MySQL hydra1 was used only for running YCSB. The data was sharded using client side hashing which evenly distributes data among MySQL servers. Performance tuning was done for the MySQL server by following the tips given in the MySQL server website.

i. 80% of the RAM of each node was allocated to the MySQL server's memory buffer pool. Having larger buffer pool size caches the record in the physical memory which reduces the disk I/O operation.

ii. The memory buffer pool was split into 3 buffer instance. Splitting up the buffer pool is needed for larger memory buffer pools and this would increase the concurrency in query execution.

iii. The read and write IO threads were configured to 32 each. These background threads are increased from its default value 4 to provide scalability.

iv. The log file size was configured as 1.5 GB, larger file reduces the number of flushes to the disk which reduces the disk I/O operations.

v. The updates are written to log file and flushed to disk every second. According to MySQL documents the updates has to be written to log file and flushed to disk for every transaction to be ACID complaint but it was traded off to accomplish scalability.

3.4 MongoDB Configuration

The MongoDB version 2.2.3 was configured with five mongod servers, three config servers and one mongos server. The mongos server was configured on hydra1 which is also shared by ycsb. There wasn't much explicit performance tuning done for MongoDB by itself had different approach like using memory buffer instead assigning dedicated main memory. Other performance tuning tip like using a random valued field as shard key for the collection was inherently exist since YCSB use hashed key. After loading the data we observed that the chunk count in all the mongod servers is close to same. The chunk size and balancing threshold was not modified while configuring.

3.5 YCSB Workload setup

YCSB version 0.1.4 was run on hydra1 server which was also used to run Hadoop's name node, HBase's hmaster, MongoDB's mongos server. However, we ensured that the server was not too loaded. Also the server application mentioned before was started whenever it is needed, for example while running HBase benchmark MongoDB application will be stopped. YCSB source code was downloaded and built locally. Some changes were made to the YCSB code to improve the performance and exception handling. In HBase client module code was added to pre create the table and regions before loading the data. MongoDB was causing type conversion exception in scan benchmarking, the code was fixed. MySQL client module was modified to ignore the record already exist exception just to make the data loading part easy and

to avoid keeping track where the last loading attempt ended/terminated abruptly. Some extra logging was added in MySQL module to capture the data shown in Figure 4.1 & 4.2.

3.6 Schema design

A table called ‘usertable’ with 10 columns and a key column was used for benchmarking. Based on the database the table schema details are provided below. Each column is about 100 bytes of data and the key is about 40 bytes. The row key was used as the sharding key in all the databases. The key was sequentially generated and hashed using Java API and prefixed with a string ‘user’.

user1000003234229993965

Figure 3.1 A sample key generated by YCSB

For benchmarking, tables or collection was created in the database. YCSB was used to load the data. YCSB creates a record with random bytes of data and inserts to the table. In HBase a table was created with a key and a column family. Each row has ten column qualifiers in the column family. As already said the regions for the table were pre created. The region in the HBase is set of records maintained with a particular range of keys. The regions were distributed all along the hosts equally by HBase itself. In sharded MySQL, a table with same name was created in all five MySQL server. The table had one primary key and ten columns of varchar type. The MySQL records were assigned to a MySQL server using application logic in YCSB. The key was hashed and modulo divided to find the MySQL server to assign the data, that way YCSB knows which MySQL server to communicate while querying or updating. In MongoDB, a collection with a shard key and ten fields were created. The data was spread across all mongod servers equally by the MongoDB.

The databases were benchmarked for three different workloads. Each workload was run for 20 minutes. The workloads are read intensive, write intensive and scan. In the read intensive workload the key to be queried was randomly chosen and queries the whole record from the database. The write intensive workload chooses key randomly and updates a single field in that record. The scan workload chooses a key randomly and gets a set of records whose keys value is greater than the specified key. The scan workload has a parameter to set how many records have to be fetched. In all of the workloads, uniform distribution is used to choose the key. For all of the benchmarking maximum 10 threads were used. Increasing the number of threads only increases the latency and gives no improvement in throughput. About 72 GB of data was generated considering each row as 1 KB of data.

Three different workloads were used in this experiment. For all the workloads the key to query was randomly decided under random distribution. The latency was calculated only using database API's duration to complete. Internal operational are not included in the latency period.

3.7 Environmental performance tuning

As suggested by the HBase performance tuning, the swap memory was disabled during all the benchmarks. Swapping memory will increase the disk IO operations which may degrade the performance. Earlier the cluster was running with 100 Mbps network switch, during which the performance of the cluster was very poor. Because of this the Gigabit network switch was installed which provided certain amount of improvement in the performance. There is no comparative performance results provided here to show the difference, but throughput measurement was given in Table 3.2.

Network switch	Throughput in MBPS
100 Mbps	94.11
1 GBPS	842.95

Table 3.2 Network bandwidth differences with old and new network switch

Three of the machines used for benchmarking had RAID level 5 configured. Although RAID 0 is suggested by HBase performance tuning, RAID 5 was used since those nodes were shared with other experiments in the research group. Also it may show if there is any perform difference between HBase and other databases because of the RAID level. The disk benchmarking results were provided Figure 3.2. The benchmarking was done using "dd" linux command by reading and writing 2 GB of data from/to the disk. These tests were run for three times and the average is given here. Most of the machines were configured with Xen kernels, since it is suspected hypervisor kernel could degrade the performance, the kernels were switched to normal kernels.

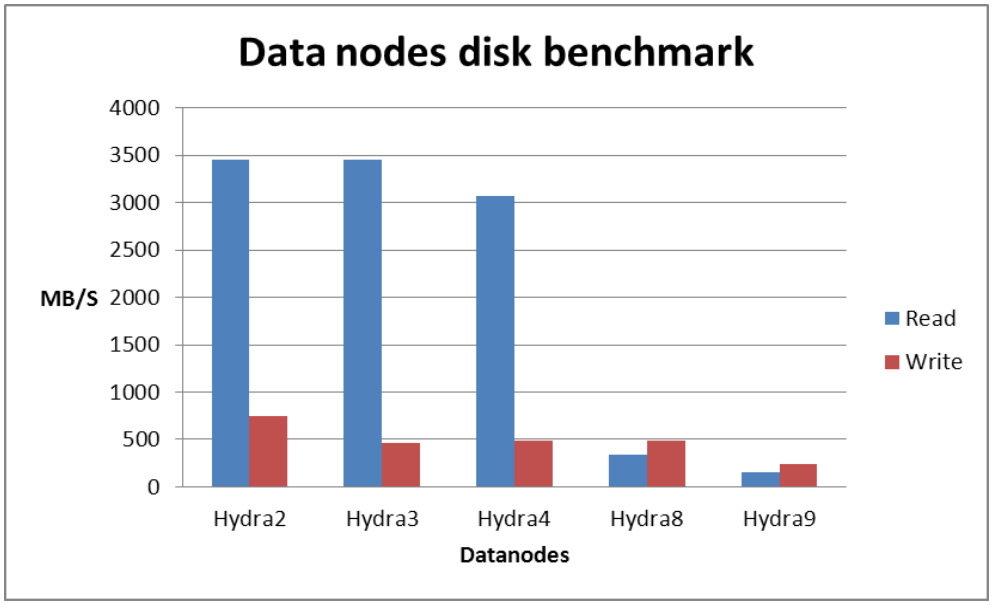

Figure 3.2 Data nodes disk benchmark

Chapter 4 Results

Here are the benchmark results and the individual node performances given for read, write and scan benchmarking with three databases HBase, MongoDB, sharded MySQL. Each workload was run for 20 minutes and for the same time average latency and average throughput was calculated and plotted here. The results provided here contain the maximum throughput achievable with the environment. Since there were bottleneck nodes in the cluster, the database's throughput did not scale up regardless of how many client threads were used. Hence for all the benchmarks only 10 client threads were used. Having more than 10 client thread only increases the latency while there is no improvement in throughput. The average latency and throughput was calculated for every 10 seconds. The benchmarks were performed with 72 GB of data which is more than all the nodes can hold in the main memory. The replication was disabled in all of the databases as this benchmark only concentrates on the performance.

The data nodes performance is measured using the metric cpu IO wait percentage 'cpu_wio' which is percentage of time the cpu waits for IO operation to complete. This metric was captured from ganglia from each of the data node for the time window when the benchmark was run and it was captured right after the benchmark was completed, so the percentage was not averaged.

In summary, the key observation of this test were,

- HBase read latency is higher than MongoDB and sharded MySQL and HBase's update latency is lower than other two. HBase is optimized for writes so this behavior is understandable.

- The scan performance of HBase is relatively well and MongoDB's scanning ability was very poor.

- When assigning the shards to data nodes, none of the database discussed here considered the capability of the node. It evenly spreads the data in all data nodes. The lesser capable nodes to struggles to compete with the other nodes mostly in disk operations and acts as a bottleneck node. The case discusses below explains this and the same behavior is observed in all of the workloads benchmarked.

Figure 4.1 shows the percentage of request which had latencies greater than 5 milliseconds for each data node captured for sharded MySQL read benchmarking. The count of read requests with considered latency is high from hydra9 & hydra8. Nodes hydra2 and hydra3 have relatively lesser percentage and there is only one such request in hydra4.

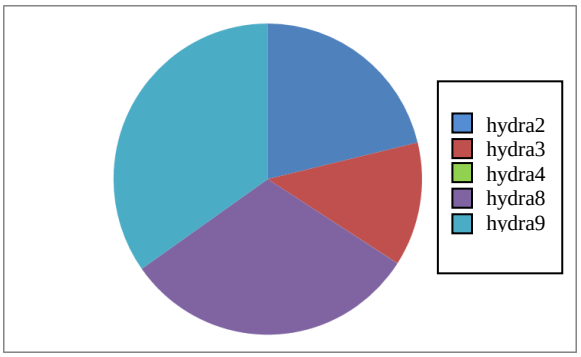

Figure 4.1 % of request had latency > 5 milliseconds of each data node

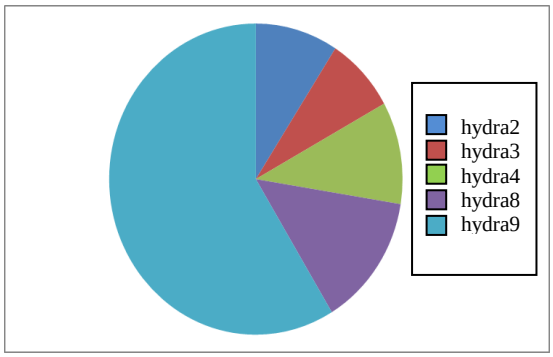

Figure 4.2 Average latency of each data node

In Figure 4.2 , the average latency (from the sampling of latency > 5 ms) of each data node is given. In that, hydra9 has the highest average latency. Though hydra8 had higher percentage of requests with considered latency its average latency is 18 ms and rest of the

machines has even lesser average latency. hydra9's average latency is about 4 times bigger than other nodes latency, which shows that the hydra9 is blocking the client threads from using the other data nodes.

4.1 Read benchmark results

Database	**Avg Throughput (**ops/sec**)**	**Avg Latency (**milliseconds**)**
HBase	276.34	36.15
MongoDB	688.67	14.51
MySQL	570.13	17.51

Table 4.1 Read benchmark Throughput & Latency

The throughputs of all the database is not enough for a cluster which has 5 datanodes. Regardless of various performance parameters tuned in the database, the throughput did not improve beyond this point which led us to use our new perspective to investigate the problem. The cpu_wio charts of all the benchmarking clearly states there is one machine hydra9 is acting as a bottleneck in the cluster. The hydra9's cpu_wio percentage is high when comparing with other nodes even though the network bytes in and out, which we use to interpret the database throughput, from each node does not differ a lot. This shows that the node is struggling to perform disk operation. The requests directed to this node has high latency which makes the client threads to wait long. So that other nodes are not leveraged very well which reduces the over all throughput. The hydra9 is acting as a bottleneck and the databases are not capable enough to understand this nature and keep the load on the bottleneck load less. Instead, the database treats all the node in the same way which is causing degrade in throughput.

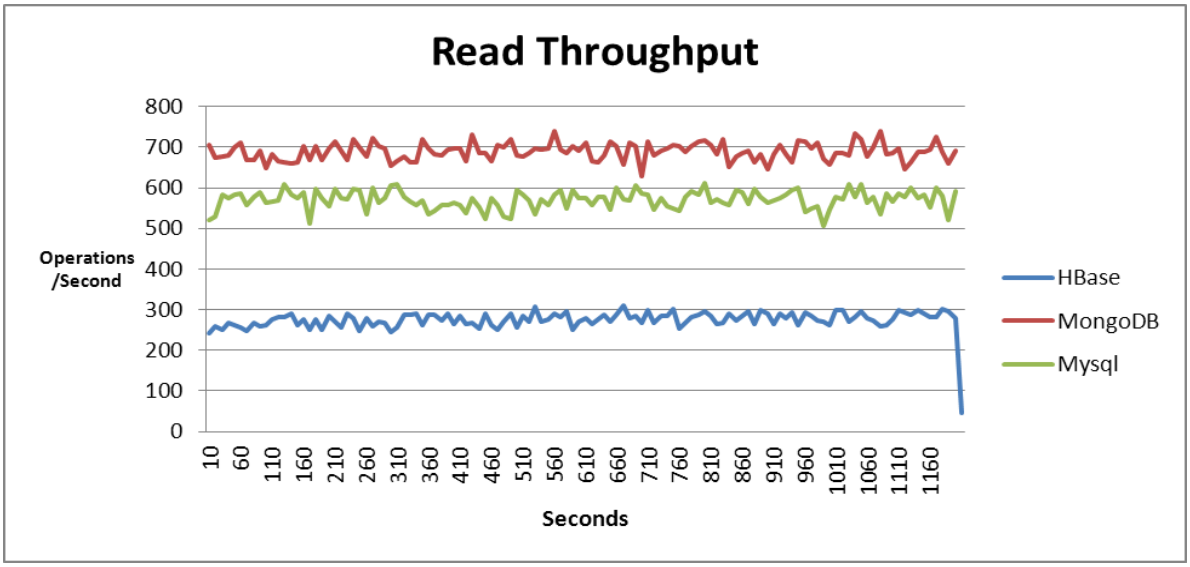

Figure 4.3 Read throughput

HBase's read latency is significantly higher than the other two databases. Since MySQL and MongoDB uses same B Tree indexing method, performance of both are close. MongoDB is able to perform very well in random read benchmarking because of using the memory buffers instead of consuming main memory and copying data in it. The cpu_wio graph given here shows the percentage of time the cpu waited for I/O operation. As described in YCSB[6] , HBase has to read and assemble records from multiple disk pages. More than that, the record has to be searched in multiple store files. However having bloomfilter enabled HBase must avoid looking into wrong files. As shown in Figure 4.4,the cpu IO wait percentage of hydra9 node for HBase read benchmark is higher than other two databases. This shows that HBase read latency is highly affected due to this bottleneck node. This also shows that HBase does more disk IO operation than the other two databases. The replication was not enabled, so this could also be a downside for HBase. If replicated, HBase would have got more than one node to fetch the data which increases the throughput.

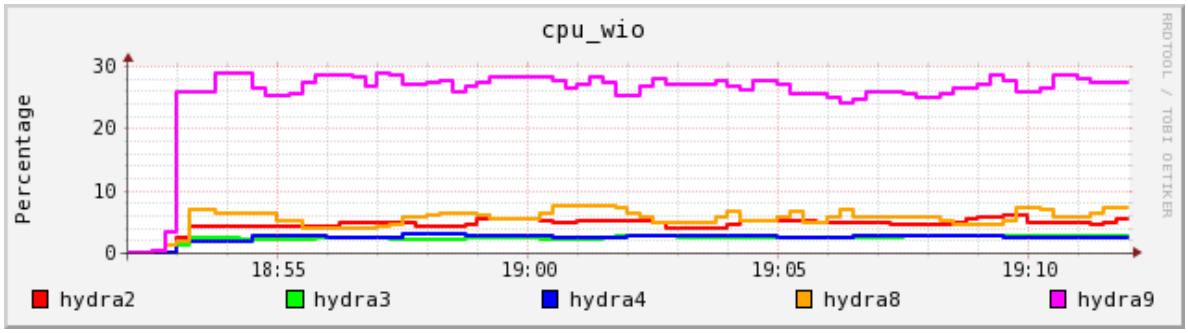

Figure 4.4 HBase cpu IO wait for read

MongoDB cpu IO wait perentage is lesser than other two databases. Memory buffering is helping it to reduce the cpu wait time. The lesser cpu IO wait is the reason for MongoDB to achieve relatively higher throughput.

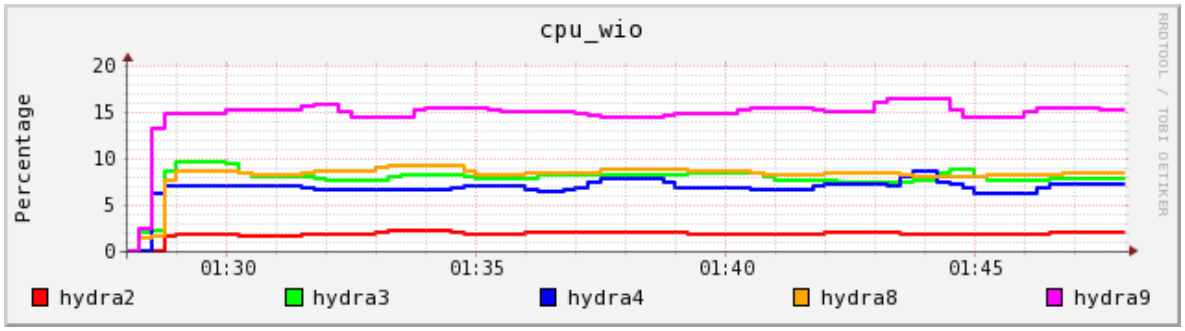

Figure 4.5 MongoDB cpu IO wait for read

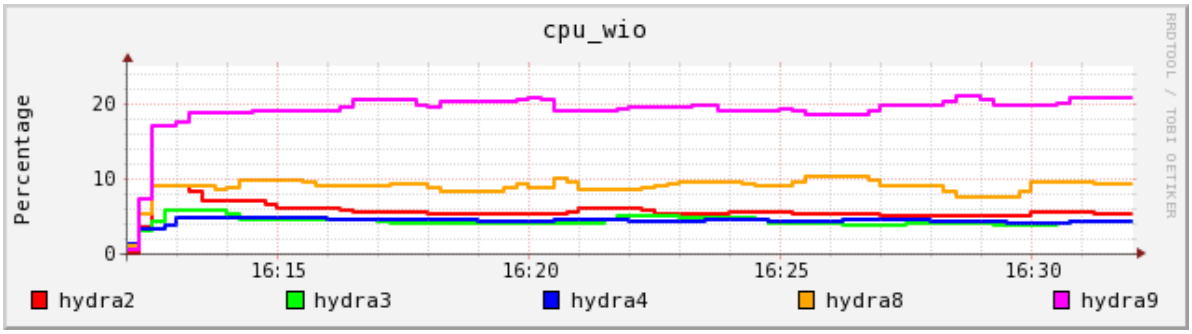

Figure 4.6 Sharded MySQL cpu IO wait for read

4.2 Write Benchmark(Update)

HBase has higher update throughput than the MongoDB and sharded MySQL, this is because the HBase writes the data in the main memory and returns but other two writes in the log file and returns. HBase gurantees the durability of the update by replicating the update. However in this benchmarking the replication was disabled. HBase has slightly high throughput when the client side buffering is enabled which collects all the updates untill the buffer is full and commits all the updates to the database later. However, since the risk of losing the data is high in this approach the client side buffering was disabled in HBase benchmarking.

Database	**Avg Throughput (**ops/sec**)**	**Avg Latency (**milliseconds**)**
HBase	11735.67	0.84
MongoDB	260.85	38.32
MySQL	504.22	19.81

Table 4.2 Write Benchmark Throughput & Latency

MongoDB has lesser throughput than MySQL. The inferred reason for this behavior is MongoDB leverages the memory caching which gives it read throughput better than MySQL but not helping in write operation. This can be noticed in the cpu wait percentage of MongoDB. MySQL consistency was traded off little bit to provide lower latency. MySQL updates were written to log files and flushed to disk every one second.

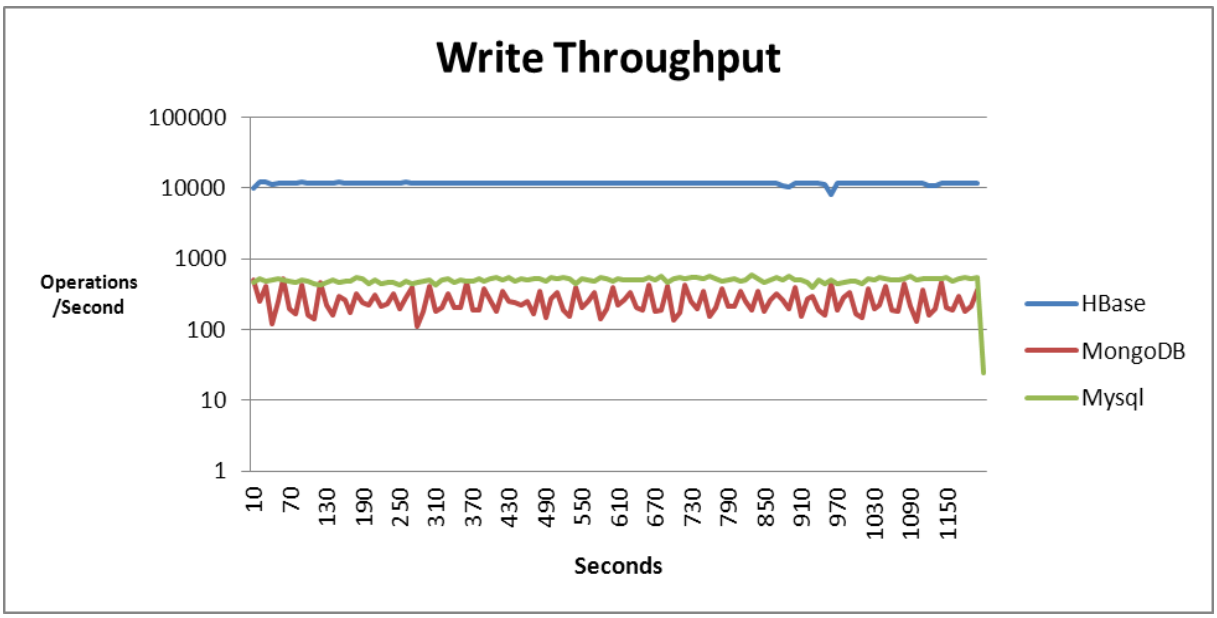

Figure 4.7 Write Benchmark Throughput

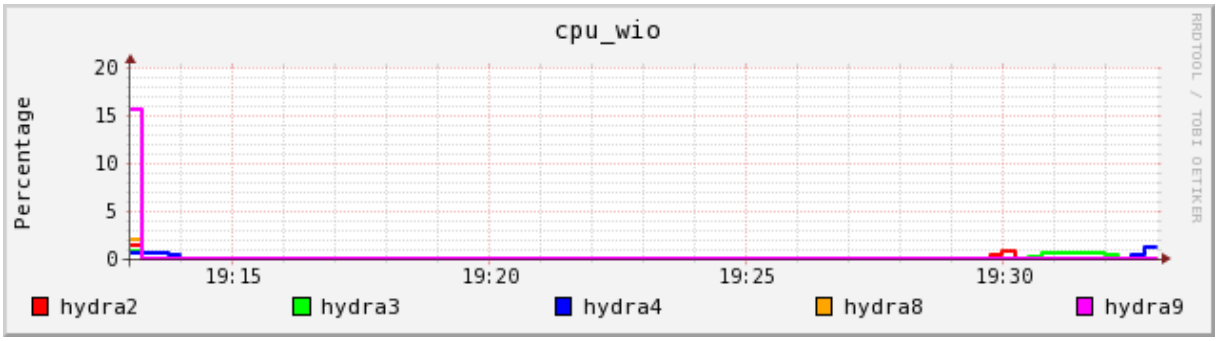

Figure 4.8 HBase cpu IO wait for write

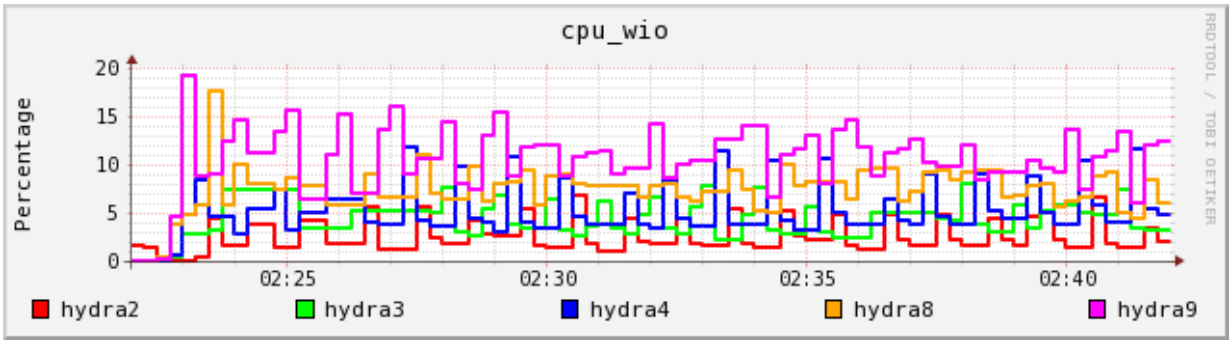

Figure 4.9 MongoDB cpu IO wait for write

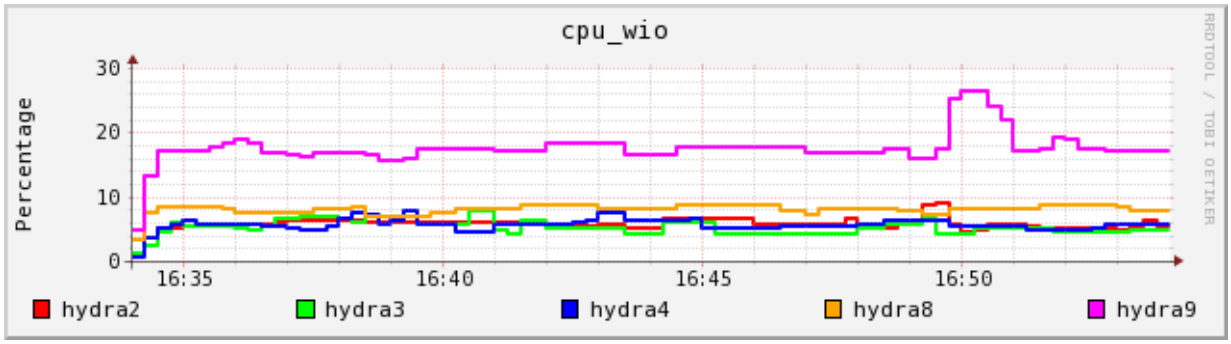

Figure 4.10 Sharded MySQL cpu IO wait for write

4.3 Scan Benchmarking

Range scan was performed using the table key with maximum of 1000 records. HBase outperforms sharded MySQL and MongoDB in scan performance. Also throughput of HBase is more consistent than other two which has throughput going up and down. MongoDB and MySQL B-Tree indexing helps it to achieve low read latency, but due to the inherent fragmentation in this approach [2], it causes high latency while scanning. Scanning was not possible with sharded MySQL architecture used for this testing, however YCSB has an implementation to scan only in one data node where the keys are greater than the chosen key. So it is not equivalent to scanning the whole table like the other two databases does.

Database	**Avg Throughput (**ops/sec**)**
HBase	42.89
MongoDB	1.27
MySQL	3.72

Table 4.3 Scan benchmark Throughput & Latency

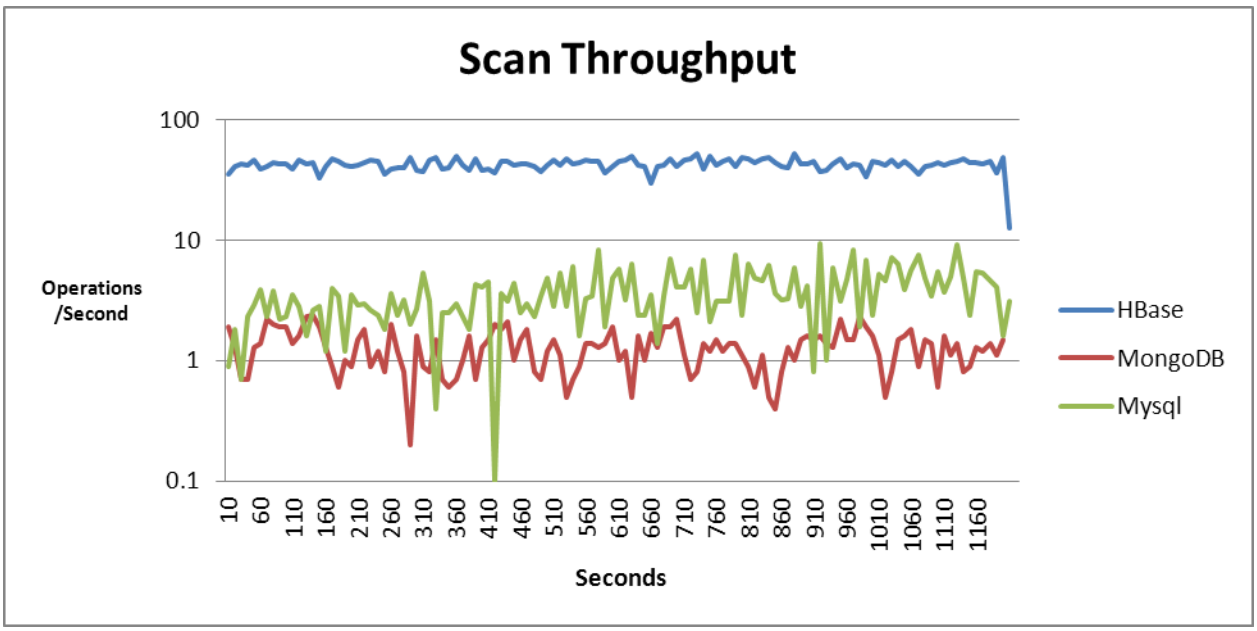

Figure 4.11 Scan Benchmark Throughput

hydra9 cpu wait I/O percentage is close to 40 % but the other machines takes only 10% – 15%. This means that HBase could actually execute more scan operation if hydra9 is not a bottleneck. For the other two database the hydra9 spends about 15% – 20 % in cpu wait I/O but still yields very less throughput.

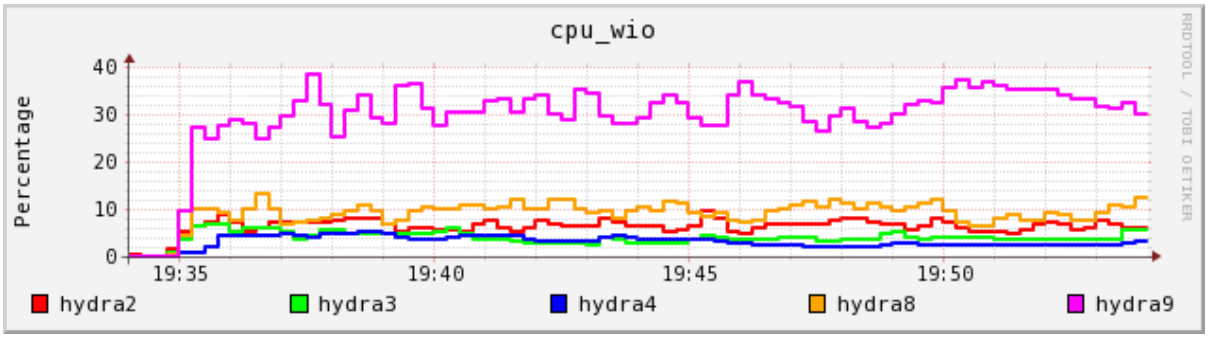

Figure 4.12 HBase cpu IO wait for scan

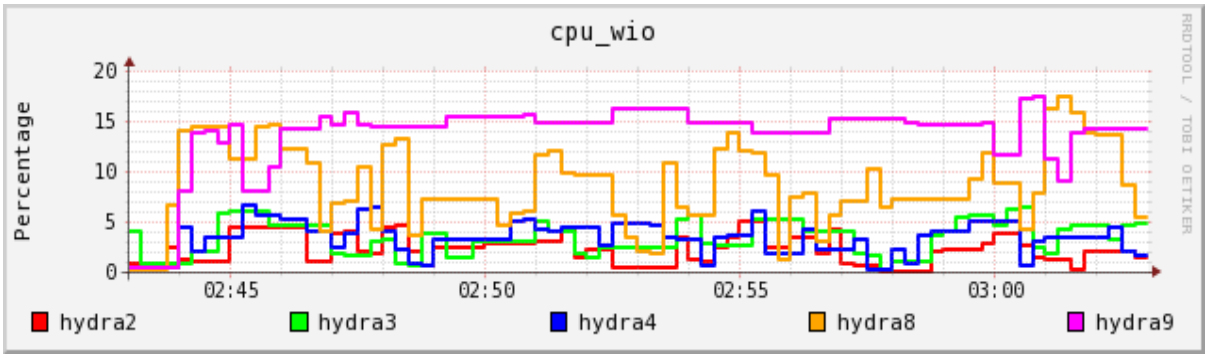

Figure 4.13 MongoDB cpu IO wait for scan

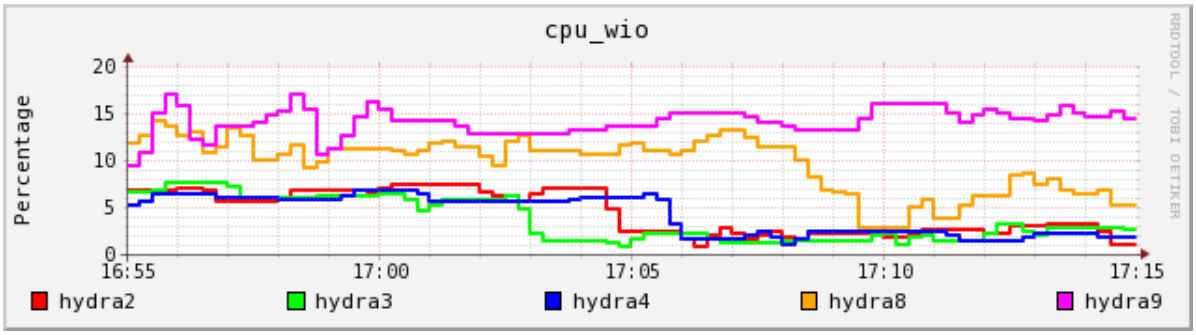

Figure 4.14 Sharded MySQL cpu IO wait for scan

4.4 Benchmarking with bottleneck nodes

To show the performance degrade when more bottleneck nodes are in the cluster, one more data node was intentionally loaded with disk operation while running the benchmark. Here the figures gives the contrast between performance of the cluster with one bottleneck node and two bottleneck nodes. The results of one bottleneck node shown here is same as the result discussed above except scan workload.

To simulate the load we used the ‘dd’ the linux command and it can be used to read/write data to/from the disk. A script which uses the ‘dd’ command will run for the configured amount of time. The script writes about 2 GB of data and deletes it and repeats until the configured duration is past. The script was started before the benchmarking and configured to beyond the end of benchmarking all the workloads.

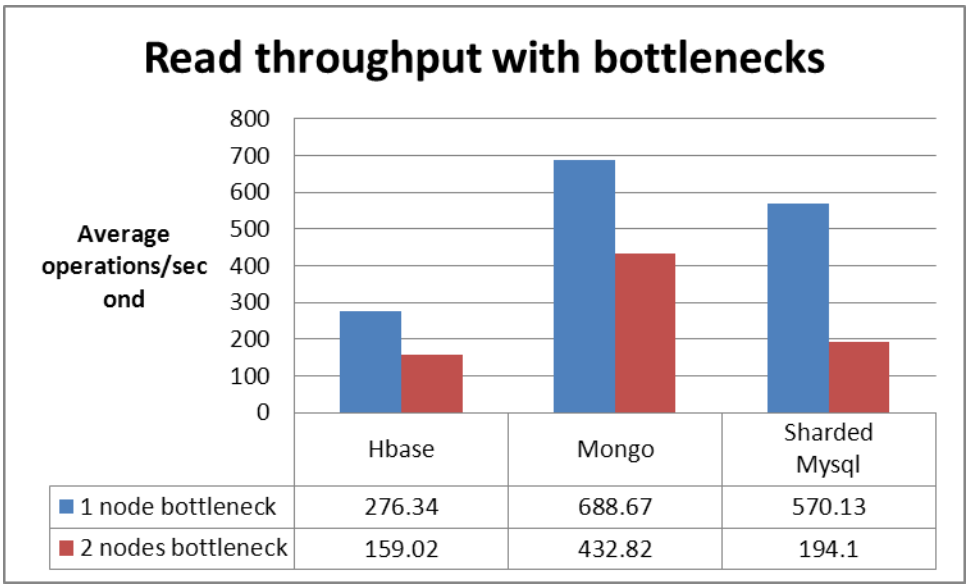

Figure 4.15 Read throughput with bottlenecks

As we have seen already HBase read performance was not good and it degrades even more when there is two bottleneck nodes. sharded MySQL degrades more than 50% of it throughput. MongoDB degrades in performance but not as bad as MySQL.

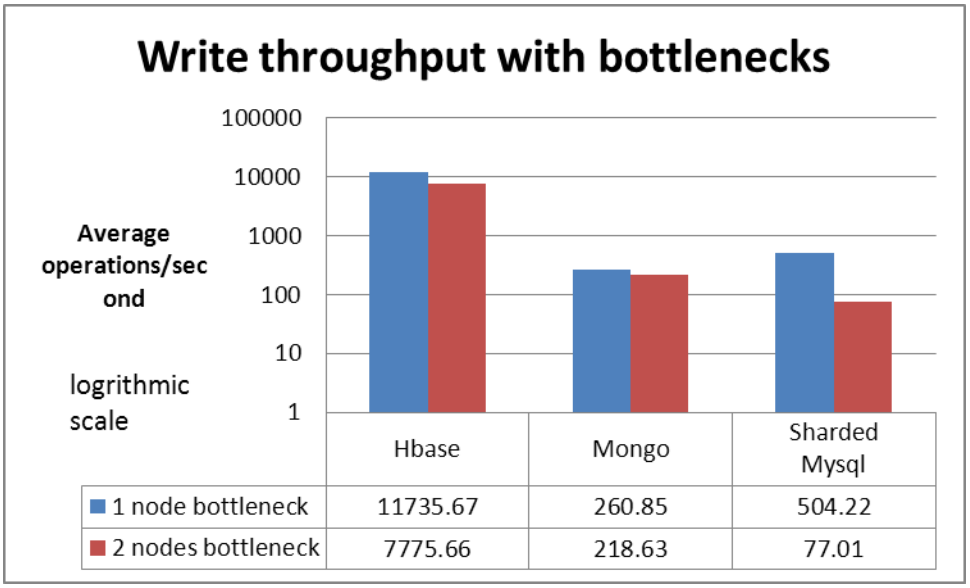

Figure 4.16 Write throughput with bottlenecks

In write benchmark, The HBase performance was degraded about 30%. It should be noted that the HBase does not sync all its update to the disk, but still there was a impact in the performance. HBase has to dump its 'Memtable' for every time it reaches 12 MB which involves disk operation and may be that's the reason why there is change seen the performance. MongoDB performance does not degrade too much. Though MongoDB did not provide a very good write performance the performance did not degrade a lot in the two bottleneck nodes situation. Unlike HBase, MongoDB syncs every write into the disk. sharded MySQL lost almost 85% of its performance. Both in read and write sharded MySQL was heavily affected by the bottleneck nodes.

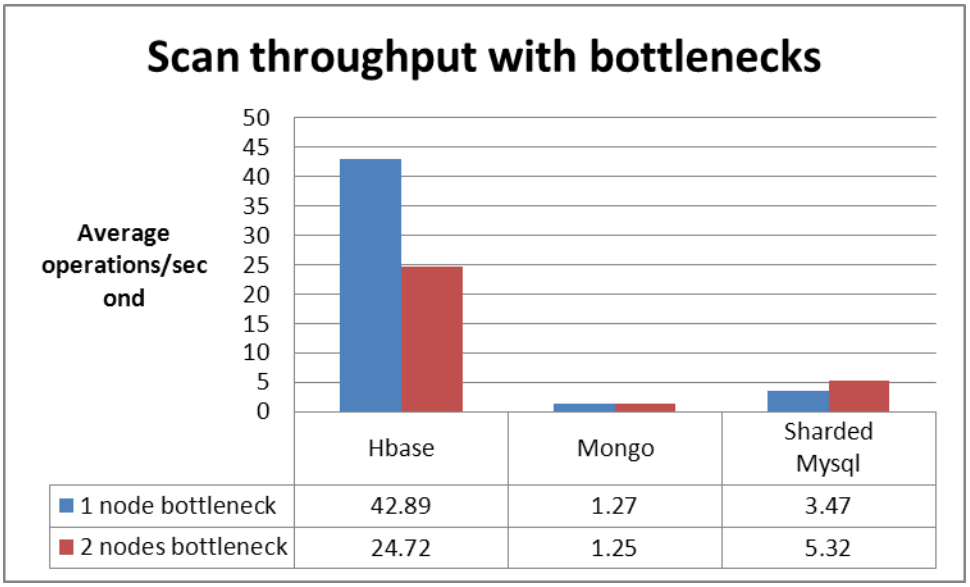

Figure 4.17 Scan throughput with bottlenecks

In scan benchmark the HBase lost its throughput about 45%, MongoDB performance was poor enough in either case. Contradictorily sharded MySQL scan perfcormance is improved in the 2 bottleneck nodes scenario. The order in which the benchmarks were run was read, write and scan. So the reason for this behavior is in 1 bottleneck node scenario more updates were performed which causes lot of buffered record to evict. During the scan more records has to be

read from disk which reduces the throughput. In case of 2 bottleneck node scenario very less updates were performed so only few records were evicted from buffer. Because of this next scan benchmark were able to read more records from buffer itself hence the throughput was higher than the previous case. When we ran the benchmarks in the order read, scan without updates the scan performance was 6.95 operations /second.

Chapter 5 Implication

In summary, we ran three different workloads to benchmark the performance of three different NoSQL databases and provided the results with individual data nodes performance. We have used heterogeneous data nodes and demonstrated how the current NoSQL database version's performance is affected by such configuration. In this section we discuss the observation and the implications.

Observation	Implication
Higher network bandwidth is needed to get high performance from NoSQL databases.	To reduce this dependency little bit, data could be transmitted in compressed format.
Disk IO affects the performance of NoSQL databases.	Having large main memory or solid state drives (SSD) in the data nodes will help increase the performance.
MongoDB read performance is better than other two databases.	MongoDB leverages the main memory buffering and achieves better random reads. So MongoDB can be used where high read bandwidth is needed.
HBase read performance is low.	HBase spends some time looking into multiple store files. Even though bloom filter is enabled it did not yield significant improvement in performance. HBase record lookup in the store files could be improved.
HBase write performance is better than other two databases. However, there is risk of losing data if the system or HBase crashes. But HBase ensures durability by replicating the data to more than one node.	HBase writes the updates in the main memory and returns. Hence it could achieve high write performance. So HBase can be used for the solution which required high write bandwidth.

HBase scan performance is relatively stable and high.	HBase can be used where high bandwidth of sequential reads is needed.
All three NoSQL database performances are affected by bottleneck nodes.	**i. NoSQL databases should consider the capability of data nodes and decide how much data to be assigned to that system.** **ii. The NoSQL databases should have the ability to detect bottleneck nodes and move the data shards dynamically from affected node to other nodes which performs well and have space for new data.** **iii. Else, if the bottleneck node is detected, then the NoSQL database can use that node only to store least accessed data. For this, the database must have the intelligence to classify data sets based on the frequency of access.**
With the existing benchmarking tool YCSB, there is no easy way to measure the latency and performance of individual nodes.	Need to extend YCSB to provide individual nodes throughput and latency statistics.

Table 5.1 Observation and Implication

Chapter 6 Related works

Michael stonebraker and Rick Cattell in the article “10 Rules for Scalable Performance in ‘Simple Operation’ Datastores” [5] provides 10 rules to consider while choosing a NoSQL database. This article discusses about NoSQL databases in many dimensions like performance, maintainability, ease of use, features to be sacrificed and few more. These rules are conceptual and can be used as a check list to filter databases at a high level. But there is not quantitative analysis provided for any specific database which this thesis concentrates.

Rick Cattell provides comprehensive details about various cloud data stores in the paper “Scalable SQL and NoSQL Data Stores” [6]. The paper discusses about the data model, consistency guarantees, replication and partitioning details of NoSQL and SQL databases and contrasts the differences. Conceptual comparison is nice to initially choose a few databases but to know if the database can render the performance expected can be only verified by benchmarking and that was not the scope of this paper.

In the paper “Benchmarking Cloud Serving Systems with YCSB” [2] discusses the need of benchmarking tool for NoSQL database and discuss their about the architecture and design of their new open source tool YCSB. In the same report they also provide the performance comparison for HBase, Cassandra, sharded MySQL and PNUTS. The scope of this paper is only within performance and elasticity and leaves scalability and availability research for future work. Our thesis is mainly based on this tool and we show that how this tool can be extended in the future.

"YCSB++ : Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores" [3] paper is very close to this thesis. As the name suggested it's an extended version of YCSB. They concentrate on advanced functionality like table week consistency, table pre splitting, bulk loading, access control and analyze their performance. In that paper they also discuss the new feature that the YCSB++ has like parallel testing which is running YCSB from multiple nodes and *performance monitoring feature* which is similar to the approach used in this thesis. In their performance monitoring they collect the cluster performance statistics through software like Ganglia, and even NoSQL database application metrics and store it in a centralized place so that these metrics can be used for performance debugging. But the functionality this thesis concentrates, analysis of collected metrics and observation and implication discussed here is different from YCSB++.

There are many other benchmarks reports available in the internet [29, 30]. Also there are few case studies[33] given in the HBase web page which concentrates on identifying performance degraders and solutions. However, the utilization of the nodes and their impact in performance was not analyzed in any of them which this thesis considers as one of the important factor and incorporated it.

Chapter 7 Conclusion & Future works

In this thesis we argue that measuring overall performance of NoSQL databases are not the end of performance analysis. The individual nodes performance is very important for the whole database performance. So the performance of individual nodes must be paid attention too. We have used three databases HBase, MongoDB and sharded MySQL for benchmarking and analyzed it with individual nodes performance metrics. Another argument of this thesis is the performance of NoSQL database not only depends on the configuration of database itself but also depends on the capability of nodes in the database cluster. In this thesis, we have shown how the performance of the database degrades by having bottleneck nodes in it. The current NoSQL balances the cluster by assigning even amount of data to the nodes in it. However, this is not always efficient if the cluster have nodes which highly vary by its capability. The thesis argues that the capability of the node should also be considered while assigning the data to it. This thesis also shows an approach to analyze the performance of individual nodes and find the bottleneck nodes in the cluster.

7.1 Future works

In future, a tool can be developed or YCSB can be extended to provide latency and throughput of individual data node of the database cluster. As like the chart provided in the MySQL latency analysis, statistics can be provided to understand which node is performing well and which node is behaving as a bottleneck. However every database should have supporting API to find out the data node associated with the record. Such tool would help finding the bottleneck nodes so that those can be fine-tuned or removed before going to production.

A NoSQL database could be altered to be sensitive to the performance of the individual nodes and balance the data according to that. The NoSQL databases should have the ability to dynamically control the load that is placed on a particular node. An advanced configuration parameter can be added to the databases which provide the percentage of load a data node should take. If the threshold is exceeded the corresponding data node should give up data to other capable nodes until the configured load percentage is met.

In this thesis, only the performance of the NoSQL databases was concentrated. Other qualities like availability, replication and elasticity were not tested. In future these qualities can also be tested and find out how individual nodes participates to provide such qualities. Especially the replication, which requires more than one writes and elasticity where the database cluster has to spread the data to new node and balance in short period of time.

Bibliography

[1] 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. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan and Russell Sears. : Benchmarking Cloud Serving Systems with YCSB. *In SoCC 2010*

[3] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs and Billie Rinaldi. YCSB++ : Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. *In SOCC 2011*

[4] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, HansArno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni PNUTS: Yahoo!'s Hosted Data Serving Platform. *In ACM VLDB '08, August 2430, 2008*

[5] Michael Stonebraker and Rick Cattell. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. *In communications of the ACM june 2011 vol. 54 no. 6 p 72 to p 80 doi :10.1145/1953122.1953144*

[6] Rick Cattell. Scalable SQL and NoSQL data stores. *In SIGMOD Record 39(4): 12-27(2010)*

[7] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). *In ACM VLDB '07, September 23-28, 2007*

[8] Dhruba Borthakur, Joydeep Sen Sarma, Jonathan Gray, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. *In SIGMOD '11, June 12?–16, 2011*

[9] Dorin Carstoiu, Elena Lepadatu, Mihai Gaspar. HBase - non SQL Database, Performances Evaluation. *In International Journal of Advancements in Computing Technology Volume 2, Number 5, December 2010*

[10] Evan Elias. tumblr. Massively Sharded MySQL. *In Velocity Europe 2011*

[11] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos and Nectarios Koziris. Distributed Indexing of Web Scale Datasets for the Cloud. *In MDAC '10, April 26, 2010*

[12] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? – Part 2: Benchmarking Results. *In 3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007*

[13] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. The Hadoop Distributed File System

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. The Hadoop Distributed File System. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *In SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*

[16] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system. *In Proceeding SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*

[17] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg. HStore: A HighPerformance, Distributed Main Memory Transaction Processing System. *In ACM. VLDB '08, August 24-30, 2008*

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *In OSDI '04: 6th Symposium on Operating Systems Design and Implementation*

[19] Matthew L. Massie , Brent N. Chun , David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *In Parallel Computing 30 (2004) 817–840*

[20] Umar Farooq Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, Sean Robertson. Elastic Scale-out for Partition-Based Database Systems. *In Proceeding ICDEW '12 Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops Pages 281-288.*

[21] Robin Hecht, Stefan Jablonski. NoSQL Evaluation: A Use Case Oriented Survey. *In Cloud and Service Computing (CSC), 2011 International Conference on 12-14 Dec. 2011*

[22] Craig Franke, Samuel Morin, Artem Chebotko, John Abraham, and Pearl Brazier. Distributed SemanticWeb Data Management in HBase and MySQL Cluster. *In Cloud Computing (CLOUD), 2011 IEEE International Conference on 4-9 July 2011 105-112*

[23] HBase - http://hbase.apache.org

[24] Hadoop - http://hadoop.apache.org

[25] The InnoDB Storage Engine

http://dev.mysql.com/doc/refman/5.6/en/innodb-storage-engine.html

[26] MongoDB BSON format

http://docs.mongodb.org/manual/reference/glossary/#term-bson

[27] Ganglia documentation - http://ganglia.sourceforge.net/

[28] RRDtool - http://oss.oetiker.ch/rrdtool/

[29] Hypertable vs. HBase Performance Evaluation II

http://hypertable.com/index.php/why_hypertable/hypertable_vs_hbase_2

[30] HBase performance testing at hstack - http://hstack.org/hbase-performance-testing

[31] Derek. Understanding Disk I/O - when should you be worried? http://blog.scoutapp.com/articles/2011/02/10/understanding-disk-i-o-when-should-you-be-worried February 2010

[32] HBase performance tuning - http://hbase.apache.org/book/performance.html

[33] Apache HBase Case Studies - http://hbase.apache.org/book/casestudies.html

Abstract

PERFORMANCE ANALYSIS OF SCALABLE SQL AND NOSQL DATABASES: A QUANTITATIVE APPROACH

by

HARISH BALASUBRAMANIAN

May 2014

Advisor: Dr.Weisong Shi

Major: Computer Science

Degree: Master of Science

Benchmarking is a common method in evaluating and choosing a NoSQL database. There are already lots of benchmarking reports available in internet and research papers. Most of the benchmark reports measure the database performance only by overall throughput and latency. This is an adequate performance analysis but need not to be the end. We define some new perspectives which also need to be considered during NoSQL performance analysis. We have demonstrated this approach by benchmarking HBase, MongoDB and sharded MySQL using YCSB. Based on the results we observe that NoSQL databases do not consider the capability of the data nodes while assigning data to it. And these databases' performance is seriously affected by the bottleneck nodes and the databases are not attempting to resolve this bottleneck situation automatically.

Autobiographical Statement

Harish Balasubramanian

Harish Balasubramanian received B.E in Computer Science & Engineering degree from Anna University in 2006. He is currently a Master's Candidate in Computer Science Department, Wayne State University, Detroit, MI. His research interest includes NoSQL databases and performance engineering.