

UNIVERSITY OF NIGERIA, NSUKKA



**SCHOOL OF POSTGRADUATE STUDIES
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRONIC ENGINEERING
OPTION: DIGITAL ELECTRONICS AND COMPUTER
SPECIALIZATION**

**STUDY AND COMPARISON OF MODERN
NOSQL MODELS FOR LARGE SCALED
WEBSITES**

BY

NWANKWO DEREK. A.

PG/M.ENG./09/51111

APPROVAL

This presentation titled the STUDY AND COMPARISON OF MODERN NOSQL MODELS FOR LARGE SCALED WEBSITES has been assessed and approved by Postgraduate Studies Committees of the Department of Digital Computer and Electronics Engineering of Electronic Engineering University of Nigeria Nsukka.

.....
Prof. Oparaku

Project Supervisor

.....
Prof. Oparaku

H.O.D. Electronic Engineering

.....
Name of Internal Examiner

.....
Dean Faculty of Engineering

CERTIFICATION

This is to certify that this research titled “STUDY AND COMPARISON OF MODERN NOSQL MODELS FOR LARGE SCALED WEBSITES” carried out by Nwankwo Derek Amarachi with registration number UNN/PG/M.ENG./09/51111 was under my supervision in the department of electronic engineering, University of Nigeria Nsukka.

.....

Name of Supervisor

.....

Signature

.....

Date

.....

Name of Head of Department

.....

Signature

.....

Date

DEDICATION

This research is first dedicated to God Almighty in Heaven for his life grace upon my life. It is also dedicated to Revs. Drs. John and Mercy Egwuonwu.

ACKNOWLEDGEMENT

I am very grateful to my Supervisor Professor Oparaku who is also my H.O.D for his help and assistance. I also acknowledge Prof. Osuagwu who laid my foundation in software engineering.

I appreciate all my Lecturers in this school who has contributed in one way or the other in helping me acquire useful knowledge.

Finally I am grateful to the almighty God for his mercy endures forever. I thank him for giving me the wisdom and resources to make this research. Glory be to God.

ABSTRACT

With the rise of internet and the explosion of data sources, more and more companies are facing new challenges that were previously very rare. Those problems concern the storage and the usability of set of data so big and often growing so fast that the usual tools were no more adapted. Moreover the multiplication of data sources and types lead to the problem of storing data that should be considered as unstructured in fixed and structured data models provided by Relational Database Management System. Those problematic lead companies and open source communities to build new tools to face those new challenges. They are known as NoSQL databases. This project titled THE STUDY AND COMPARISON OF NOSQL MODELS FOR LARGE SCALED WEBSITES takes a practical approach at studying these NoSQL models, finding their scalability properties and investigating performances. Matlab diagrams are used for comparison. These results as well as CAP theorem were used to make suggestions and use cases to help people know which will best suit their website. It also solves some problems encountered in some of these database models.

TABLE OF CONTENT

Title Page.....	i
Approval Page	ii
Certification	iii
Dedication	iv
Acknowledgement	v
Abstract	vi
Table Of Contents	viii
List Of Figures	xi
List Of Tables	xiii
Chapter One	
Introduction	PAGE
1.1. Background Of Study	1
1.2. Aims And Objectives	2
1.3. Scope Of Study	2
1.4. Justification Of Study	2
1.5. Constraints And Limitations	2
Chapter Two	
Literature Review	3
2.1 Structured Query Language	3
2.1.1. Why NoSQL Models?	4
2.1.2. Problems That A NoSQL Database Tries To Solve	4
2.1.3. SQL And NoSQL Scaling	6
2.2. Partitioning	7
2.3. Storage Layout	14
2.3.1 Row-Based Storage Layout	15
2.3.2 Columnar Storage Layout	15
2.3.3. Columnar Storage Layout with Locality Groups	15
2.3.4. Log Structured Merge Trees	15
2.4. Storage Distribution	20
2.5 The CAP Theorem	25
2.6. Distribute The Computation	26
2.7. Study Of Some Selected Database	28
2.7.1. HBase	28
2.7.2 Cassandra	34
2.7.3. MongoDB	38
2.9.3. CAP Approach	42

2.7.4.	Riak	44
2.7.5.	Scalaris	46
2.8.	YCSB	50
2.9.	Catalyst 2950 Switch	51
2.10.	Matlab Description	51
2.11.	Matrix And Array Operations	54
Chapter Three	Methodology	58
3.1.	Parameters/Properties Being Measured	58
3.1.1.	Performance	58
3.1.2.	Scalability	58
3.1.3.	Stability	58
3.2.	Materials Used	59
3.2.1.	Software Configuration Settings	59
3.2.2.	Cassandra	59
3.2.3.	Hbase	60
3.3.4.	MongoDB	60
3.3.	Data Set	61
3.3.1.	Creating The Mydata Keyspace And The Two Column Families	64
3.3.2	Createdata.php. Script	65
3.3.3.	Readwrite100million.php Script	66
3.3.4.	Readwrite100million.php Code	67
3.3.5.	Readwrite200million.php Script	69
3.3.6.	The Result Format	70
3.4.	Program Flow	71
3.5	Readwrite100million.php	72
3.6.	Network Configuration	73
3.7.	Network Diagrams	74
3.8.	Solving Load Balancing Problem In NoSQL	75
3.9	Experiment Showing The Effect Of Load Balancing	76
3.10.	Application And Result On My Cluster	80
3.11.	Microsoft System Center Virtual Machine Manager (VMM)	81
3.12.	Running The Virtual Machine Converter Software	81

Chapter Four	Results Analysis And Discussion	83
4.1.	Results And Calculations	83
4.2.	Math Lab Diagrams	85
4.3.	Stabilization Times Of The Database Models	88
4.4.	Scalability	89
4.5.	Performance Measurement	91
4.6.	Comparison With Results From Ecole Polytechnique De Louvain Computer Engineering Dept.	93
4.7.	Results Of Physical And Virtual Nodes	97
4.8.	Performance And Stability Results	100
4.9.	Results Analysis	101
Chapter Five	Conclusion And Recommendation	105
5.1.	Use Cases	105
5.2.	Website With Medium To High Traffic	105
5.3.	File Storage In The Cloud	107
5.4.	Conclusions	109
5.5.	Further Work	110
A	References	112
B	Appendix	116

LIST OF FIGURES

2.1.	Consistent Hashing – Initial Situation	11
2.2.	Consistent Hashing – Situation After Node Joining And Departure	11
2.3.	Consistent Hashing – Virtual Nodes Example	12
2.4.	Consistent Hashing – Example With Virtual Nodes And Replicated Data	13
2.5.	Row-based	16
2.6.	Columnar-based	16
2.7.	Columnar With Locality Groups	17
2.8.	Data Layout	17
2.9.	Read Path	17
2.10.	Read Path With Bloom Filters	17
2.11.	Write Path	17
2.12.	Flushing Of Memtables	18
2.13.	Compaction Of SS Tables	18
2.14.	Storage Layout – Memtables And SStables In Bigtable	19
2.15.	Storage Layout – Copy-on-modify in CouchDB	20
2.16.	Consistent Hashing Illustration	21
2.17.	Architecture Of A Google File System Cluster	24
2.18.	Architecture Of The Meta Data In A Bigtable Cluster	25
2.19.	Architecture Of A Hbase Cluster	28
2.20.	Requests Made Into A Newly Bootstrapped Hbase Cluster	30
2.21.	Requests Made Into An Hbase Cluster Running In Production To Get A Database Entry	31
2.22.	Hbase's Data Model Representation	32
2.23.	Cassandra Data Model : The Two Rows Are Identified By Key1 And Key2	35
2.24.	Cassandra Data Model: The Columnfamily Model Extended To Supercolumn	36
2.25.	MongoDB Not Sharded With Replication Architecture	39
2.26.	MongoDB Sharded Cluster With Replication	40
2.27.	MongoDB's Data Model Representation	41
2.28.	Riak's Data Model Representation	44
2.29.	Voldemort Architecture	48

2.30.	Matlab Screen	52
2.31.	Line Plots	55
2.32.	Sine Function	56
2.33.	Line Specification	56
2.34.	Cosine And Sin	57
3.1.	The Data And Timer Diagram	63
3.2.	The Writer Diagram	64
3.3.	Program flow diagram (createdata.php And insertdata.php) ..	71
3.4.	Program flow diagram (readwrite100million.php)	72
3.5.	6 Nodes Cluster	74
3.6.	6 -12 Nodes Cluster	74
3.7.	12 - 24 Nodes Cluster	75
3.8.	24 Nodes On A Ring Cluster	75
3.9.	4 Hashed Node On A Cluster	76
3.10	The result of application	79
3.11	Installation Screen	81
3.12.	Run Screen	82
3.13.	VMware Converter	82
4.1	From 6 To 12 Nodes Results	83
4.2.	From 12 To 24 Nodes Results	84
4.3.	Cassandra (6 – 12 Nodes)	85
4.4.	Hbase (6 – 12 Nodes)	85
4.5.	Mongodb (6 – 12 Nodes)	86
4.6.	Mongodb (12 – 24 Nodes)	86
4.7.	Cassandra (12 – 24 Nodes)	87
4.8.	Hbase (12 – 24 Nodes)	87
4.9.	Compared Cassandra (6 - 12 nodes)	94
4.10.	Compared Cassandra (12 - 24 nodes)	94
4.11.	Compared Hbase (6 - 12 nodes)	95
4.12.	Compared Hbase (12 - 24 nodes)	95
4.13.	Compared Mongo (6 - 12 nodes)	96
4.14.	Compared MongoDB (12 - 24 nodes)	96
4.15.	Results Of Physical And Virtual Nodes	97
4.16.	Physical (6 – 12 nodes)	98
4.17.	Virtual (6 – 12 nodes)	98

4.18.	Comparison between Virtual & Physical (6 – 12 nodes)	99
-------	--	----

LIST OF TABLES

3.1.	Data	62
3.2.	Timer Table	63
3.3.	Writer Table	63
3.4.	Result From Time Column Family	70
3.5.	Network Diagram	73
4.1.	Stabilization Time (In Minute Lower Is Better)	88
4.2.	Hbase Has The Best Scalability Property	90
4.3.	Performance In Seconds	93

CHAPTER ONE

INTRODUCTION

1.1. Background of study

With the rise of Internet and the explosion of data sources, more and more companies are facing new challenges that were previously very rare. Those problems concern the storage and the usability of set of data so big and often growing so fast that the usual tools are no more adaptable. Moreover, the multiplication of data sources and types lead to the problem of storing data that should be considered as unstructured in fixed and structured data models provided by Relational Database Management System. These problems lead companies and open source communities to building of new tools to face these new challenges. These new models are known as NoSQL databases.

The world of NoSQL databases is very interesting but also quite complex due to several factors:

- NoSQL databases are evolving rapidly, therefore critics and analysis can be obsolete very fast.
- There is a profusion of NoSQL databases and each of them has its own specifications.
- The various NoSQL databases have reached different level of maturity and it is not always trivial to determine if a given database or a subset of its functionalities is ready for production.

1.2. AIMS AND OBJECTIVES

Due to new database models emerging every day, there is a problem of determining which will best fit a particular type of website. Therefore this project practically examines the most common NoSQL models in other to find their scalability properties, stability and performances as well as look into their CAP properties according to the CAP theorems to enable the development of a use case or advisory that can suit any particular website. This project also solves some problems encountered in some of these NoSQL models.

1.3. SCOPE OF STUDY

This project concentrates more on the three most common NoSQL models used today which are the Cassandra, the Hbase and the MongoDB. These are perfect NoSQL models and are mostly used by large scaled sites in the world today.

1.4. JUSTIFICATION OF STUDY

The work done here will be a guide to a lot of companies whose database are growing beyond what RDBMS or SQL. It also suits companies moving into NoSQL models just like facebook, Twiter, Amazon and a lot others moved into NoSQL models. It will also be of great help to companies who are already using these models but are facing some of the problems solved in this thesis. It will actually help them speed up there database processing, increase efficiency and also save cost.

1.5. CONSTRAINTS AND LIMITATIONS

Some of these large scaled sites have servers that number into hundreds, some even thousands but due to cost and availability of resources I was limited to experimenting with only 30 nodes / systems.

CHAPTER TWO

Literature Review

2.1. Structured Query Language (SQL([84](#)))

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s. This version, initially called **SEQUEL** (*Structured English Query Language*), was designed to manipulate and retrieve data stored in IBM's original quasi-relational database management system, System R, which a group at IBM San Jose Research Laboratory had developed during the 1970s. The acronym SEQUEL was later changed to SQL because "SEQUEL" was a trademark of the UK-based Hawker Siddeley aircraft company.

The first *Relational Database Management System* (RDBMS) was RDMS, developed at MIT in the early 1970s, soon followed by Ingres, developed in 1974 at U.C. Berkeley. Ingres implemented a query language known as QUEL, which was later supplanted in the marketplace by SQL.

In the late 1970s, Relational Software, Inc. (now Oracle Corporation) saw the potential of the concepts described by Codd, Chamberlin, and Boyce and developed their own SQL-based RDBMS with aspirations of selling it to the U.S. Navy, Central Intelligence Agency, and other U.S. government agencies. In June 1979, Relational Software, Inc. introduced the first commercially available implementation of SQL, Oracle V2 (Version2) for VAX computers. *Oracle V2* beat IBM's August release of the System/38 RDBMS to market by a few weeks.

After testing SQL at customer test sites to determine the usefulness and practicality of the system, IBM began developing commercial products based on their System R prototype including System/38, SQL/DS, and DB2, which were commercially available in 1979, 1981, and 1983, respectively.

2.1.1. WHY NOSQL MODELS?

With the rise of Internet and the explosion of data sources, more and more companies are facing new challenges that were previously very rare. Those problems concern the storage and the usability of set of data so big and often growing so fast that the usual tools were no more adapted. Moreover the multiplication of data sources and types lead to the problem of storing data that should be considered as unstructured in fixed and structured data models provided by Relational Database Management System. Those problematic lead companies and open source communities to build new tools to face those new challenges. They are known as NOSQL databases.

2.1.2 PROBLEMS THAT A NOSQL DATABASE TRIES TO SOLVE

The main problems that a NOSQL aims to solve typically revolve around issues of scale. When data no longer fits on a single MySQL server or when a single machine can no longer handle the query load, some strategy for sharding and replication is required.

The pitch behind most NOSQL databases like Cassandra, HBase, Voldemort and others is that because they were designed from the ground up to be distributed and to handle large data volumes, they can provide some combination of the following benefits that a simple installation of MySQL or Postgres can't easily offer:

1. Automatic sharding of data. New data gets automatically assigned to the appropriate node.
2. Automatic replication of data. Multiple nodes each store a copy of the data, up to a certain configured replication factor.
3. Schema-less data for simpler migrations. Schema changes for large tables can take a long time and lock the tables, blocking any writes. A database with only

a loosely defined schema (like Cassandra and HBase's column families) or none at all in key/value stores should make this easier.

4. Automatic scalability by adding new nodes. Adding new nodes automatically re-partition the data for load balancing purposes.
5. Multiple nodes that can accept writes. Unlike a standard MySQL master/slave setup, multiple nodes in a NOSQL database can accept updates, thereby supporting much higher query throughput.

A few other key/value stores that are often lumped into the NOSQL category, like Redis and Tokyo Cabinet, aim less to provide distributed scalability but instead optimize for high-performance lookups at the cost of no longer supporting relational queries.

In practice, the level and reliability of support for each of these benefits varies from system to system. Facebook, Friendfeed, Ning, and other companies have demonstrated that scaling capabilities can often be built within the application layer on top of standard relational databases like MySQL; moreover, failures with Cassandra at Digg [1] and the decision not to use Cassandra for the primary data store at Twitter [2] suggest that the MySQL-based systems may, at least for now, be more robust than many of the newer, unvetted systems. The outcome of Facebook's decision to use HBase for its new messages product may change this landscape.

2.1.3. SQL AND NOSQL SCALING

We are all using the Web 2.0. With its huge amount of data going into pentabytes (10^{15} bytes), the relational data model is no longer appropriate because it cannot handle it. A possible solution to this performance problem is to scale the database. This can be done in two different ways.

SQL databases mainly use vertical scalability, meaning that a performance advantage is only enabled by increasing the performance of the database server itself. This is also called *up scaling*. However, huge amounts of data cannot be handled by just one server that is more powerful because the hardware (e.g. card slots, operating system) is sometime limited.

On the other side the horizontal scaling allows the addition of different distributed database servers for a better performance. This is also called *out scaling*. Note: Distribution alone does not automatically increase the performance of the database! If the database is still managed by one server in the network, then the performance, e.g. number of writes/reads per seconds, does not improve. A horizontal scalability is only given if the computing of queries is also distributed, so that the performance really improves.

2.2. Partitioning

Assuming that data in large scale systems exceeds the capacity of a single machine and should also be replicated to ensure reliability and allow scaling measures such as load-balancing, ways of partitioning the data of such a system have to be thought about. Depending on the size of the system and the other factors like dynamism (e. g. how often and dynamically storage nodes may join and leave) there are different approaches to this issue:

Memory Caches

Like memcached [2] can be seen as partitioned—though transient— in-memory databases as they replicate most frequently requested parts of a database to main memory, rapidly deliver this data to clients and therefore disburden database servers significantly. In the case of memcached the memory cache consists of an array of processes with an assigned amount of memory that can be launched on several machines in a network and are made known to an application via configuration. The memcached protocol [3] implementations are available in key-/value-store API. It stores objects placed under a key into the cache by hashing that key against the configured memcached-instances. If a memcached process does not respond, most API implementations ignore the non-answering node and use the responding nodes instead which leads to an implicit rehashing of cache-objects as part of them gets hashed to a different memcached-server after a cache-miss; when the formerly non-answering node joins the memcached server array again, keys for part of the data get hashed to it again after a cache miss and objects now dedicated to that node will implicitly leave the memory of some other memcached server they have been hashed to while the node was down (as memcached applies a LRU strategy for cache cleaning and additionally allows it to specify timeouts for cache-objects). Other implementations of memory caches are available e. g. for application servers like JBoss [4].

Clustering

Clustering of database servers is another approach to partition data which strives for transparency towards clients who should not notice talking to a cluster of database servers instead of a single server. While this approach can help to scale the persistence layer of a system to a certain degree many criticize that clustering features have only been added on top of DBMSs that were not originally designed for distribution.

Separating Reads from Writes

Separating Reads from Writes means to specify one or more dedicated servers, write-operations for all or parts of the data are routed to (master(s)), as well as a number of replica-servers satisfying read requests (slaves). If the master replicates to its clients asynchronously there are no write lags but if the master crashes before completing replication to at least one client the write-operation is lost; if the master replicates writes synchronously to one slave lags the update does not get lost, but read request cannot go to any slave if strict consistency is required and furthermore write lags cannot be avoided (5)

[6] and regarding write lags StudiVz's Dennis Bemann's comments [7]. In the latter case, if the master crashes the slave with the most recent version of data can be elected as the new master. The master/slave-model works well if the read/write ratio is high. The replication of data can happen either by transfer of state (i. e. copying of the recent version of data or delta towards the former version) or by transfer of operations which are applied to the state on the slave's nodes and have to arrive in the correct order (8).

Sharding

Sharding means to partition the data in such a way that data typically requested and updated together resides on the same node and that load and storage volume is roughly even distributed among the servers (in relation to their storage volume and processing power; as an example confer the experiences of Bemann with a large German social-network on that topic [7]). Data shards may also be replicated for reasons of reliability and load-balancing and it may be either allowed to write to a dedicated replica only or to all replicas maintaining a

partition of the data. To allow such a sharding scenario there has to be a mapping between data partitions (shards) and storage nodes that are responsible for these shards. This mapping can be static or dynamic, determined by a client application, by some dedicated “mapping-service/component” or by some network infrastructure between the client application and the storage nodes. The downside of sharding scenarios is that joins between data shards are not possible, so that the client application or proxy layer inside or outside the database has to issue several requests and post process (e. g. filter, aggregate) results instead. Lipcon therefore comments that with sharding “you lose all the features that make a RDBMS useful” and that sharding “is operationally obnoxious” (9). This valuation refers to the fact that sharding originally was not designed within current RDBMSs but rather added on top. In contrast many NOSQL databases have embraced sharding as a key feature and some even provide automatic partitioning and balancing of data among nodes—as e.g. MongoDB of version 1.6 [10],[11],[12]).

In a partitioned scenario knowing how to map database objects to servers is key. An obvious approach may be a simple hashing of database-object primary keys against the set of available database nodes in the following manner: *partition = hash(o) mod n* with *o = object to hash, n = number of nodes*

As mentioned above the downside of this procedure is that at least parts of the data have to be redistributed whenever nodes leave and join. In a memory caching scenario data redistribution may happen implicitly by observing cache misses, reading data again from a database or backend system, hashing it against the currently available cache servers and let stale cache data be purged from the cache servers by some cleaning policy like Least recently used (LRU). But for persistent data stores this implicit redistribution process is not acceptable as data not present on the available nodes cannot be reconstructed [13], [6]. Hence, in a setting where nodes may join and leave at runtime (e. g. due to node crashes, temporal un attainability and maintenance work) a different approach such as consistent hashing has to be found which shall be discussed hereafter.

Consistent Hashing

The idea of consistent hashing was introduced by David Karger et al. in 1997 [14] in the context of a paper about “a family of caching protocols for distributed [sic!] networks that can be used to decrease or eliminate the occurrence of hot spots in the networks”. This family of caching protocols grounds on consistent hashing

which has been adopted in other fields than caching network protocols since 1997, e. g. in distributed hash-table implementations like Chord [15] and some memcached clients as well as in the NOSQL scene where it has been integrated into databases like Amazon's Dynamo and Project Voldemort.

"The basic idea behind the consistent hashing algorithm is to hash both objects and caches using the same hash function" blogger Tom White points out. Not only hashing objects but also machines has the advantage, that machines get an interval of the hash-function's range and adjacent machines can take over parts of the interval of their neighbors if those leave and can give parts of their own interval away if a new node joins and gets mapped to an adjacent interval. The consistent hashing approach furthermore has the advantage that client applications can calculate which node to contact in order to request or write a piece of data and there is no metadata server necessary as in systems like e. g. the Google File System (GFS) which has such a central (though clustered) metadata server that contains the mappings between storage servers and data partitions (called *chunks* in GFS; cf. [16]).

Figures below and illustrate the idea behind the consistent hashing approach. It there are three red colored nodes A, B and C and four blue colored objects 1–4 that are mapped to a hash-function's result range which is imagined and pictured as a ring. Which object is mapped to which node is determined by moving clockwise around the ring. So, objects 4 and 1 are mapped to node A object 2 to node B and object 3 to node C. When a node leaves the system, cache objects will get mapped to their adjacent node (in clockwise direction) and when a node enters the system it will get hashed onto the ring and will overtake objects. An example is depicted in figure [Situation after Node Joining and Departure] where compared to figure [Initial Situation] nodes C left and node D entered the system, so that now objects 3 and 4 will get mapped to node D. This shows that by changing the number of nodes not all objects have to be remapped to the new set of nodes but only part of the objects.

In this raw form there are still issues with this procedure: at first, the distribution of nodes on the ring is actually random as their positions are determined by a hash function and the intervals between nodes may be "unbalanced"

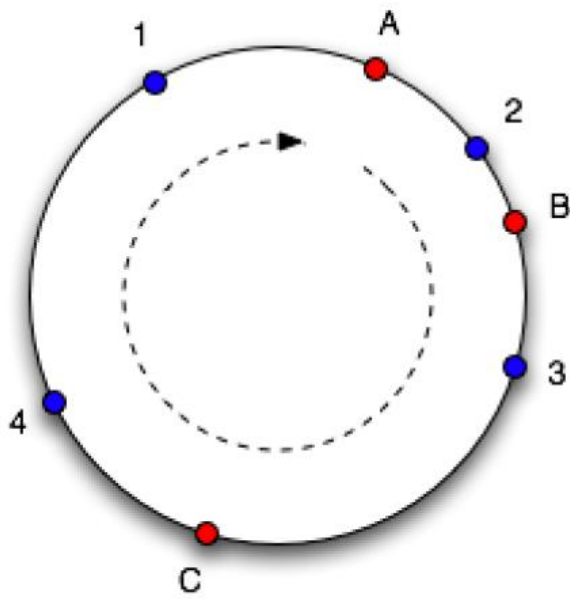


Fig 2.1: Consistent Hashing – Initial Situation (taken from [17])

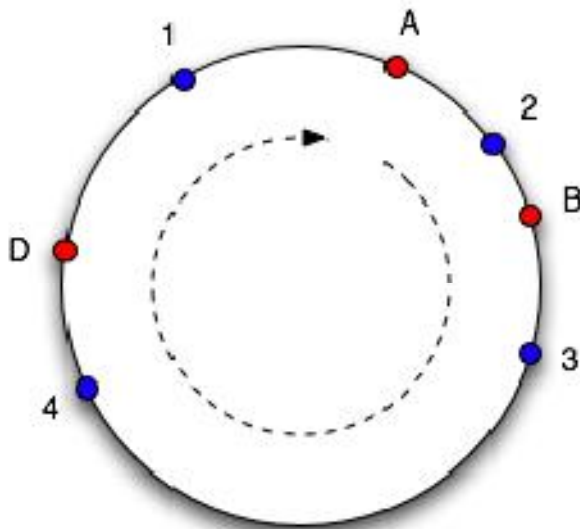


Figure 2.2: Consistent Hashing – Situation after Node Joining and Departure (taken from [18])

which in turn results in an unbalanced distribution of cache objects on these nodes (as it can already be seen in the small scenario of figure (2.2) where node D has to take cache objects from a greater interval than node A and especially node B). An approach to solve this issue is to hash a number of representatives/replicas—also called virtual nodes—for each physical node onto

the ring as an example see figure (2.3). The number of virtual nodes for a physical node can be defined individually according to its hardware capacity (CPU, memory, disk capacity) and does not have to be the same for all physical nodes. By appending e. g. a replica counter to a node's id which then gets hashed, these virtual nodes should distribute points for this node all over the ring.

In his blog post on consistent hashing Tom White has simulated the effect of adding virtual nodes in a setting where he distributes 10,000 objects across ten physical nodes. As a result, the standard deviation of the objects distribution can be dropped from 100% without virtual nodes to 50% with only 2–5 virtual nodes and to 5–10% with 500 virtual nodes per physical node (17).

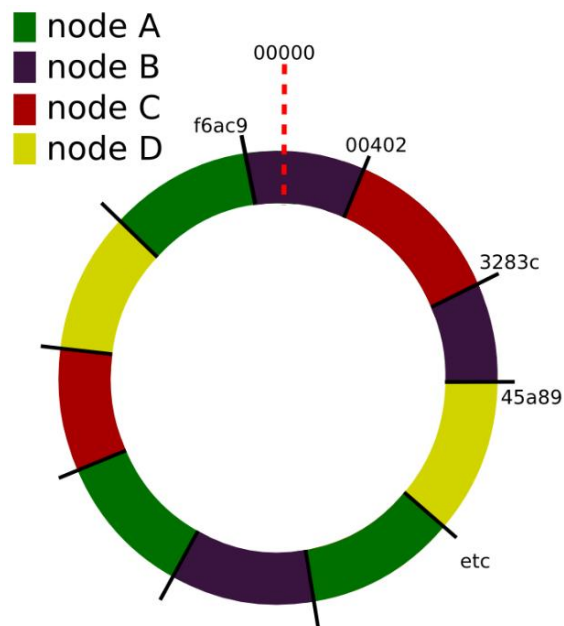


Figure 2.3: Consistent Hashing – Virtual Nodes Example
(Taken from [19])

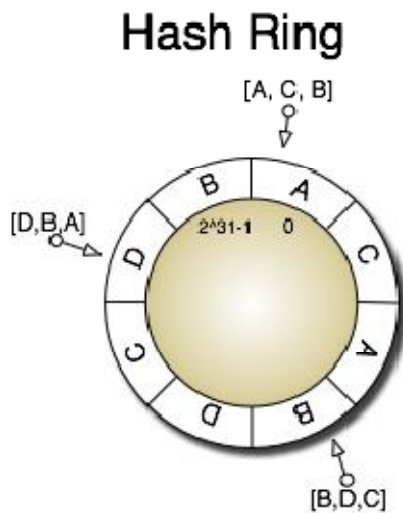


Figure 2.4: Consistent Hashing – Example with Virtual Nodes and Replicated Data (taken from [20])

When applied to persistent storages, further issues arise: if a node has left the scene, data stored on this node becomes unavailable, unless it has been replicated to other nodes before; in the opposite case of a new node joining the others, adjacent nodes are no longer responsible for some pieces of data which they still store but not get asked for anymore as the corresponding objects are no longer hashed to them by requesting clients. In order to address this issue, a replication factor (r) can be introduced. By doing so, not only the next node but the next r (physical!) nodes in clockwise direction become responsible for an object [20], [21]. Figure [Virtual Nodes and Replicated Data](#) depict such a scenario with replicated data: the uppercase letters again represent storage nodes that are – according to the idea of virtual nodes–mapped multiple times onto the ring, and the circles with arrows represent data objects which are mapped onto the ring at the depicted positions. In this example, the replication factor is three, so for every data object three physical nodes are responsible which are listed in square brackets in the figure. Introducing replicas implicates read and write operations on data partitions which will be considered in the next subsection. The above mentioned issues causing membership changes shall be discussed thereafter.

Read- and Write Operations on Partitioned Data

Introducing replicas in a partitioning scheme—besides reliability benefits—also makes it possible to spread workload for read requests that can go to any physical node responsible for a requested piece of data. In qualification, it should be stated that the possibility of load-balancing read operations does not apply to scenarios in which clients have to decide between multiple versions of a dataset and therefore have to read from a quorum of servers which in turn reduces the ability to load-balance read requests. The Project Voldemort team points out that three parameters are specifically important regarding read as well as write operations ([22])

N: The number of replicas for the data or the piece of data to be read or written.

R: The number of machines contacted in read operations.

W: The numbers of machines that have to be blocked in write operations⁵.

In the interest to provide e. g. the read-your-own-writes consistency model the following relation between the above parameters becomes necessary:

$$R + W > N$$

Regarding write operations clients have to be clear that these are neither immediately consistent nor isolated ([23]):

_ If a write operation completes without errors or exceptions a client can be sure that at least W nodes have executed the operation.

_ If the write operation fails as e. g. less than W nodes have executed it, the state of the dataset is unspecified. If at least one node has successfully written the value it will eventually become the new value on all replica nodes, given that the replicas exchange values and agree on the most recent versions of their datasets in background. If no server has been capable to write the new value, it gets lost.

Therefore, if the write operation fails, the client can only achieve a consistent state by re-issuing the write operation.

2.3. Storage Layout

In his talk on design patterns for non relational databases Todd Lipcon presents an overview of storage layouts, which determine how the disk is accessed and therefore directly implicate performance. Furthermore, the storage layout defines

which kind of data (e. g. whole rows, whole columns, subset of columns) can be read (24).

2.3.1. Row-Based Storage Layout

Row-Based Storage Layout means that a table of a relational model gets serialized as its lines are appended and flushed to disk (see figure 2.5). The advantages of this storage layout are that at first whole datasets can be read and written in a single IO operation and that secondly one has a “[good locality of access (on disk and in cache) of different columns]”. On the downside, operating on columns is expensive as a considerable amount data (in a naïve implementation all of the data) has to be read.

2.3.2. Columnar Storage Layout

Columnar Storage Layout serializes tables by appending their columns and flushing them to disk (see Fig 2.6). Therefore operations on columns are fast and cheap while operations on rows are costly and can lead to seeks in a lot or all of the columns. A typical application field for this type of storage layout is analytics where an efficient examination of columns for statistical purposes is important.

2.3.3. Columnar Storage Layout with Locality Groups

Columnar Storage Layout with Locality Groups is similar to column-based storage but adds the feature of defining so called locality groups that are groups of columns expected to be accessed together by clients. The columns of such a group may therefore be stored together and physically separated from other columns and column groups (see figure 3.10c). The idea of locality groups was introduced in Google’s Bigtable paper. It firstly describes the logical model of column-families for semantically affine or interrelated columns (cf. [CDG+06, section 2]) and later on presents the locality groups idea as a refinement where column-families can be grouped or segregated for physical storage (cf. [CDG+06, section 6]).

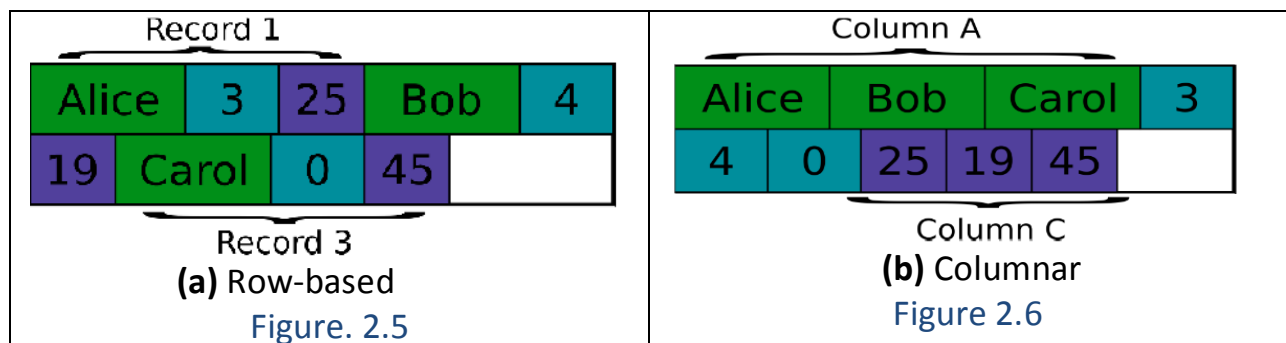
2.3.4. Log Structured Merge Trees (LSM-trees)

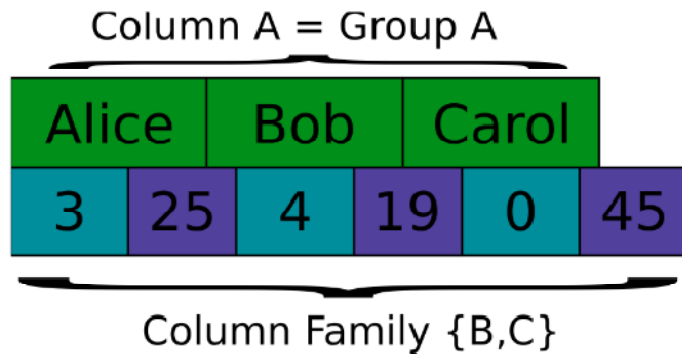
Log Structured Merge Trees in contrast to the storage layouts explained before do not describe how to serialize logical data structures (like tables, documents etc.) but how to efficiently use memory and disk storage in order to satisfy read and write requests in an efficient performance and still safely manner. The idea, brought up by O’Neil et al. in 1996 (25) is to hold chunks of data in memory (in so

called Memtables), maintaining on-disk commit-logs for these in-memory data structures and flushing the memtables to disk from time to time into so called SSTables (see [Fig 2.5 and 2.6](#)). These are immutable and get compacted over time by copying the compacted SSTable to another area of the disk while preserving the original SSTable and removing the latter after the compaction process has happened (see [figure Storage Layout – Log Structured Merge Trees](#)). The compaction is necessary as data stored in these SSTable may have changed or been deleted by clients. These data modifications are first reflected in a Memtable that later gets flushed to disk as a whole into a SSTable which may be compacted together with other SSTables already on disk. Read-requests go to the Memtable as well as the SSTables containing the requested data and return a merged view of it (see [figure 2.10](#)).

To optimize read-requests and only read the relevant SSTables bloom filters⁶ can be used (see [figure 2.10](#)). Write requests go to the Memtable as well as an on-disk commit-log synchronously (see [figure 2.11](#)).

The advantages of log structured merge trees are that memory can be utilized to satisfy read requests quickly and disk I/O gets faster as SSTables can be read sequentially because their data is not randomly distributed over the disk. LSM-trees also tolerate machine crashes as write operations not only go to memory but also (synchronously) to a commit-log by which a machine can recover from a crash. Log structured merge trees are implemented in e. g. Google's Bigtable ([\[26\]](#), sections 4, 5.3, 5.4, 6) as blogger Ricky Ho summarizes graphically in figures [Storage Layout – MemTables and SSTables in Bigtable](#).





(c) Columnar with locality groups

Figure 2.7: Storage Layout – Row-based, Columnar with/out Locality Groups
(From ([9] Presentation of 2009-06-11.)

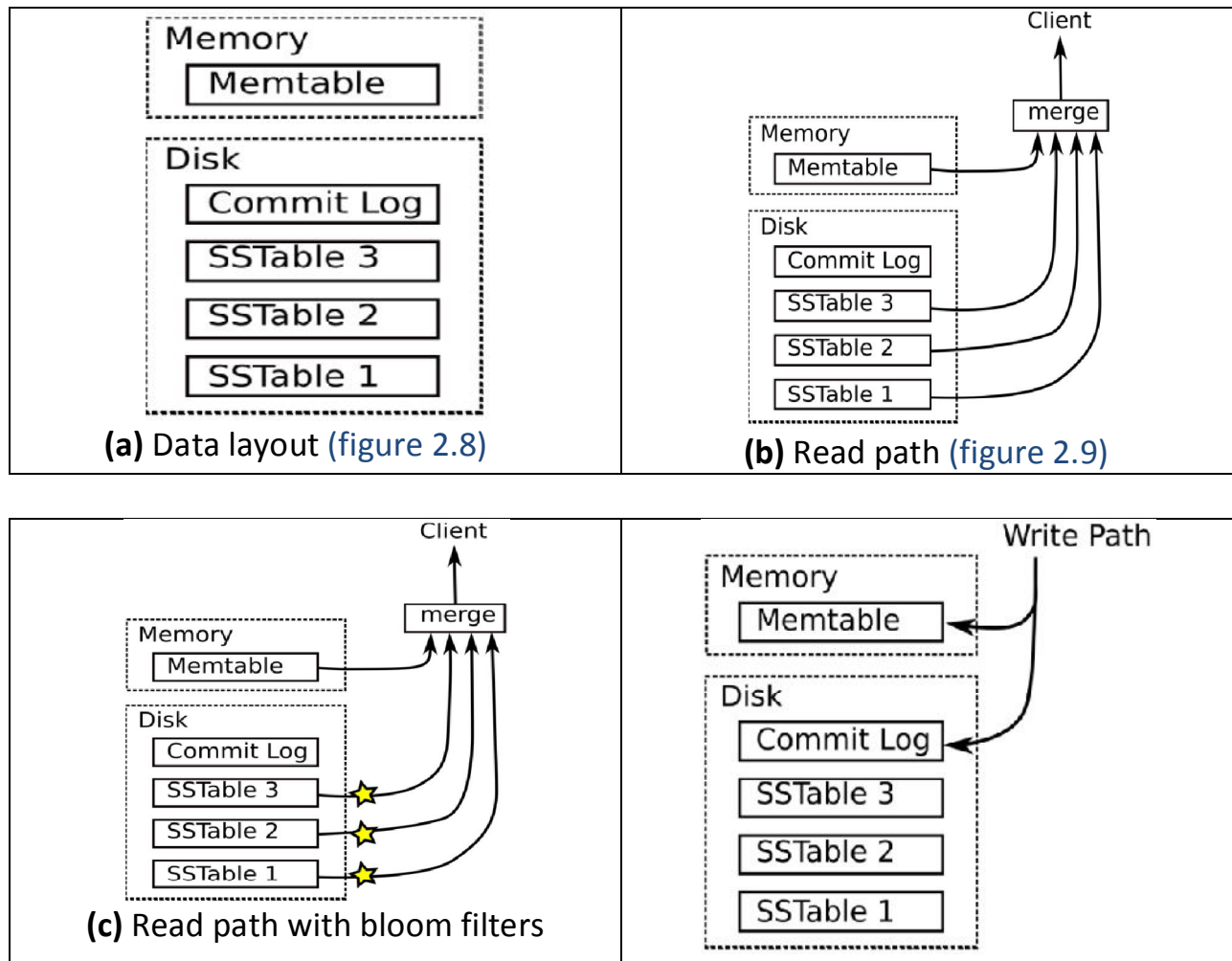
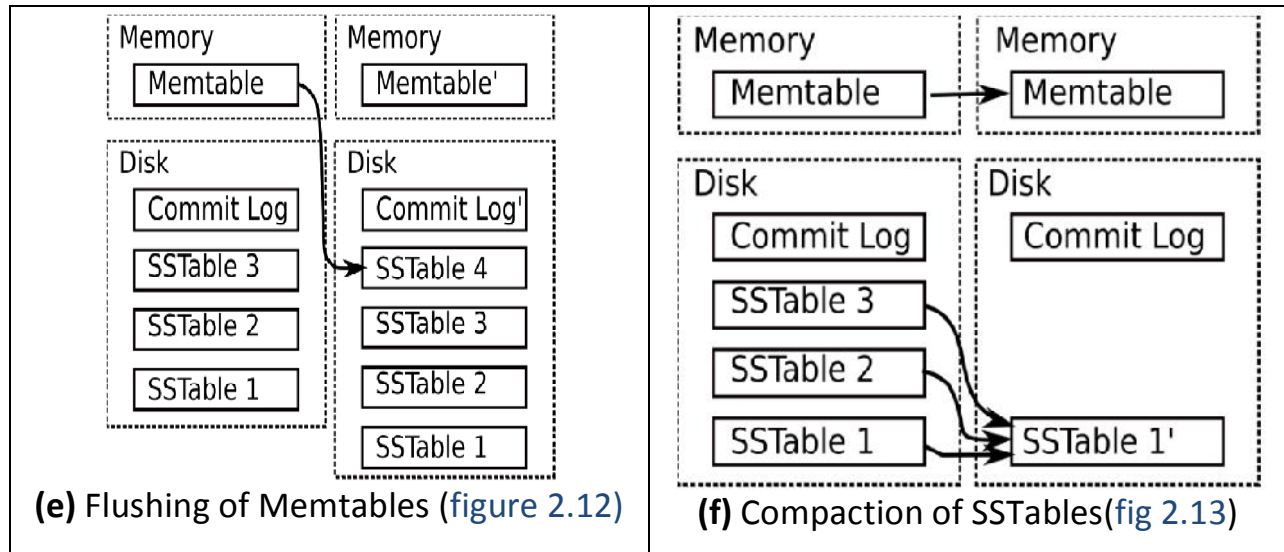


Figure 2.10

(d) Write path (figure 2.11)



Storage Layout – Log Structured Merge Trees

Regarding the discussion of on-disk or in-memory storage or a combination of both (as in LSM-trees) blogger Nati Shalom notes that it “comes down mostly to cost/GB of data and read/write performance”.

He quotes from an analysis by the Stanford University ([27]) “cost is also a function of performance”. He cites from the analysis that “if an application needs to store a large amount of data inexpensively and has a relatively low access rate”.

RAM Cloud is not the best solution but for systems with high throughput requirements a RAM-Cloud [sic!] can provide not just high performance but also energy efficiency ([28]).

Blogger Ricky Ho adds to the categorization of storage layouts above that some NOSQL databases leave the storage implementation open and allow to plug-in different kinds of it—such as Project Voldemort which allows to use e. g. a relational database, the key/value database Berkley DB, the file system or a memory hash table as storage implementations. In contrast, most NOSQL databases implement a storage system optimized to their specific characteristics. In addition to these general storage layouts and the distinction between pluggable and proprietary storage implementations most databases carry out optimizations according to their specific data model, query processing means, indexing structures etc.

As an example, the document database CouchDB provides a copy-on-modified semantic in combination with an appended file for database contents. The copy-on modified semantic means that a private copy for the user issuing the modification request is made from

the data as well as the index (a B-Tree in case of CouchDB); this allows CouchDB to provide read-your own- writes consistency for the issuer while modifications are only eventually seen by other clients. The private copy is propagated to replication nodes and—as CouchDB also supports multi-version concurrency control (MVCC)—maybe clients have to merge conflicting versions before a new version becomes valid for all clients (which is done by swapping the root pointer of storage metadata that is organized in a B-Tree as shown in [figure 2.15](#)).

CouchDB synchronously persists all updates to disk in an append-only fashion. A garbage-collection compacts the persisted data from time to time by copying the compacted file contents into a new file and not touching the old file until the new one is written correctly. This treatment allows continuous operation of the system while the garbage-collection is executed.

Ho depicts how the CouchDB index and files are affected by updates as shown in [figure 2.14](#).

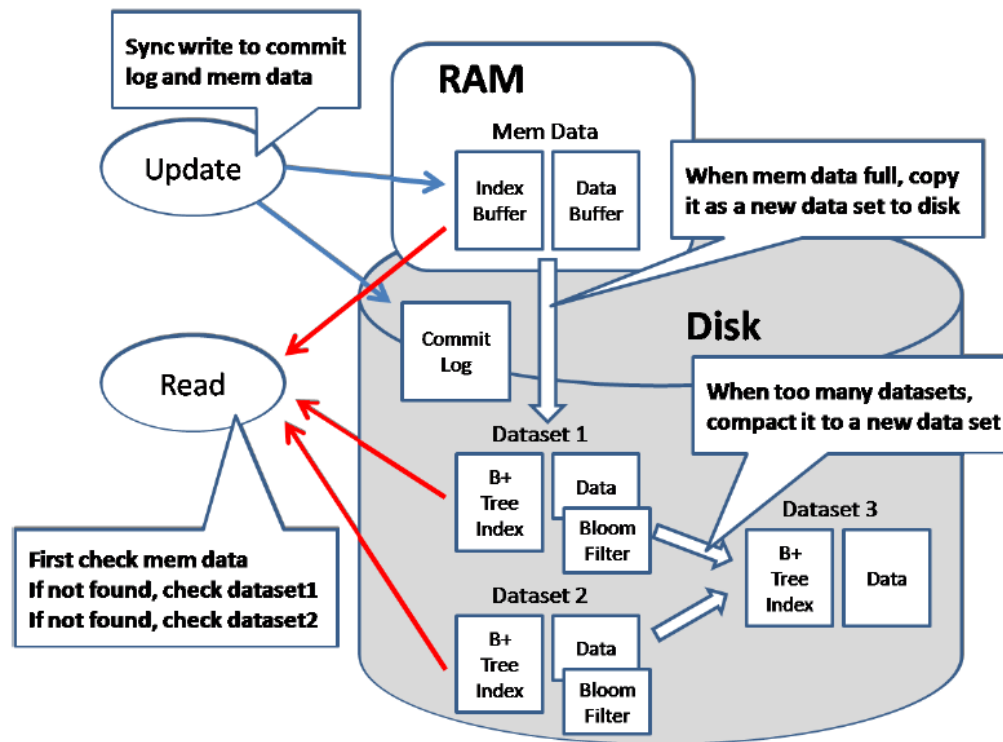
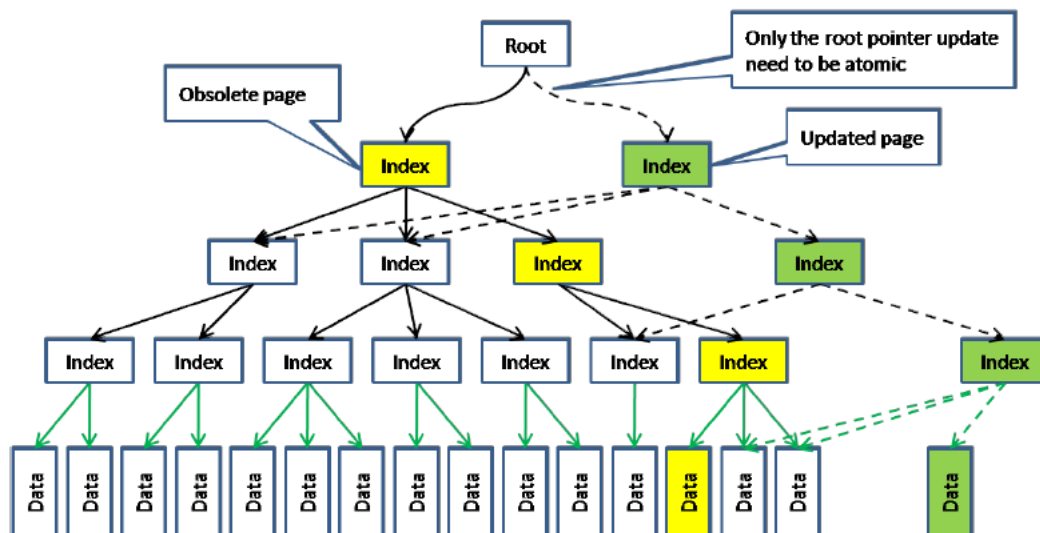


Fig 2.14: Storage Layout – MemTables and SSTables in Bigtable
(Taken from [6])



Copy on modify. Everyone sees his own copy of update

Finally the root pointer is swapped and everyone's view is updated

Yellow page becomes garbage over time.

File will be compacted periodically by copying to a different file.

Fig 2.15: Storage Layout – Copy-on-modify in CouchDB

2.4. STORAGE DISTRIBUTION

Consistent hashing based distributed storage

The method Consistent hashing [29] provides the basic functionalities of a distributed hash table, but the main advantage compared to a simple hash table is that the number of keys that must be moved from a slot to another when slots are added or removed is only of k/n , with k the number of keys and n the number of slots.

With consistent hashing, the hash space, that is the range of all the possible hash that can be produced by the hashing function, is represented as a ring. That implies that every new key can be placed on that ring after its hash has been computed. In the same way, the servers making the cluster can be placed on that ring by assigning each of them a hash code as well.

Once the servers are placed on the ring, the range of hash codes they are responsible for is defined as all the keys on the ring going from their current position to the next server on the ring. The [Figure 2.16](#) Below shows an example with 3 servers and 4 items placed on the ring. In this example the server s_1 is responsible for the keys k_1 and k_2 , s_2 is responsible for the key k_3 and s_3 is responsible for the key k_4 .

If the server s_2 dies, the ownership of its part of the ring will be assigned to the previous server on the ring, s_1 . Therefore there is only one key that changes of server in this case instead of a complete new assignation of all the keys to the remaining servers.

In practice there are two big approaches to ensure a good distribution of the data among the nodes of the cluster. Indeed, as shown before, the

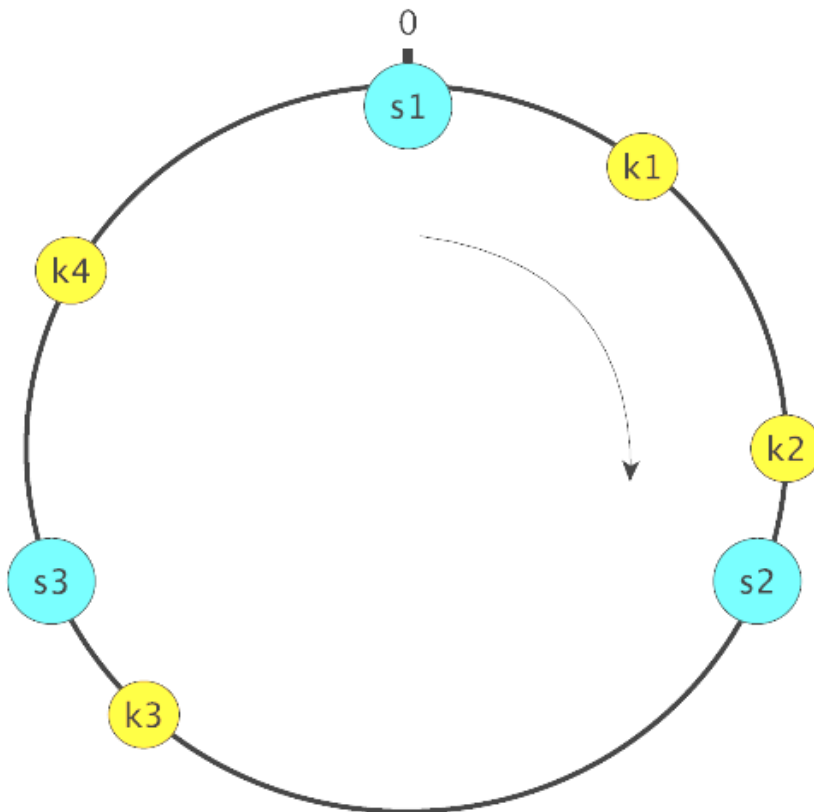


Fig 2.16: Consistent hashing illustration

Number of keys a server is responsible for depends on its position on the ring and badly chosen positions on the ring will lead to an unbalanced cluster.

The first step is dividing the hash space into equal parts by assigning n virtual servers to each of those equidistant positions. Those subdivisions of the ring are often called partitions and to ensure that the cluster is well balanced, each of the m real servers is assigned the ownership of $n=m$ partitions.

This is the way distributed systems like Amazon's Dynamo and Riak ensure that a cluster stays automatically balanced. The advantages, given that the number of virtual servers is big enough compared to the number of real servers. The advantages of using virtual servers are:

- If a real server dies, the virtual servers it was responsible for can be distributed evenly across the remaining nodes
- When a real server is added to the cluster, it can take a fair share of the load of all the other servers
- It is possible to assign a different number of virtual servers to each of the real servers, meaning that is possible to take into account the fact that some servers are less powerful than others.

The second approach to this problem is to let the administrators of the distributed storage decide where each server should be on the ring. That implies that the administrators of the database have good understanding of how the position of a server on the ring will influence its load. Cassandra [30.], [31.] is a system where it is possible to define the position¹ of each server on the ring. The advantages of this method is that if a given part of the ring is a hot-spot that generate a lot of traffic, it is easy to add a new server that will split in two the load of this exact server.

Storage distribution based on centralized directory

One of the alternatives to the consistent hashing method is to use a central authority in the cluster that monitor the servers storing the data chunks and decide how the chunks are distributed among the cluster. This implies that a client that wants to access a specific data must contact this central authority at least one time before it can contact the server storing the right chunk. In practice, this central authority can be distributed on more than one server and there are several ways to avoid making it a bottleneck. The general idea behind the distributed storage systems based on a centralized directory is to divide the data into chunks of fixed size, each of them identified by a unique ID. Then the mapping between the chunks and the servers, as well as some meta information, is stored on the central authority. In practice there are two big approaches regarding the architecture of the central authority, the main difference between them being the number of levels on which this information is stored.

The first approach is to make all the information stored by the central authority available in one step, meaning that a client that wants to locate a chunk on the cluster has only to ask for it to the central authority and gets back its answer

directly. Then it can open a connection to the server that stores the chunk to get the data it was looking for. This is the approach taken by the Google File System [32.] where the central authority is called the master and is a single process running on a single server. The fact that the master is running on a single server simplifies the architecture of the whole software and, as the master has a global knowledge of the cluster, enables it to make sophisticated decisions about chunks placement and replication.

The drawback of this approach is of course the fact that the master is a single point of failure but this can be lessened by the client's cache that store all the meta information that it previously received from the master. Nevertheless, if the master is down, a new client with an empty cache would be unable to find anything on the cluster and even long-running client would not be able to find a block if its location has changed meanwhile. The architecture of an

Google File System cluster is shown in the [Figure 2.17](#)

The central authority can store directly all the meta information of the cluster and still be distributed to be more resilient to failure. As the clients rely on this central authority to find specific data into the cluster, the information stored on it must always be consistent. To ensure that a consistent view is always offered to the clients, a synchronous replication or another protocol enforcing atomicity must be used. This is the approach taken by mongoDB [33.] where all the meta data of the cluster are stored on three servers using two phase commit to ensure meta data consistency. The fact that two phase commit is used implies that no more writes or meta data update will be possible as soon as one of the three servers is down, meaning that no more chunk splitting or balancing will be possible. But on the other hand, the advantage compared to the single process central authority, is that the clients will still be able to read the meta data from the two remaining servers and therefore most of the cluster operations will continue without any perturbations.

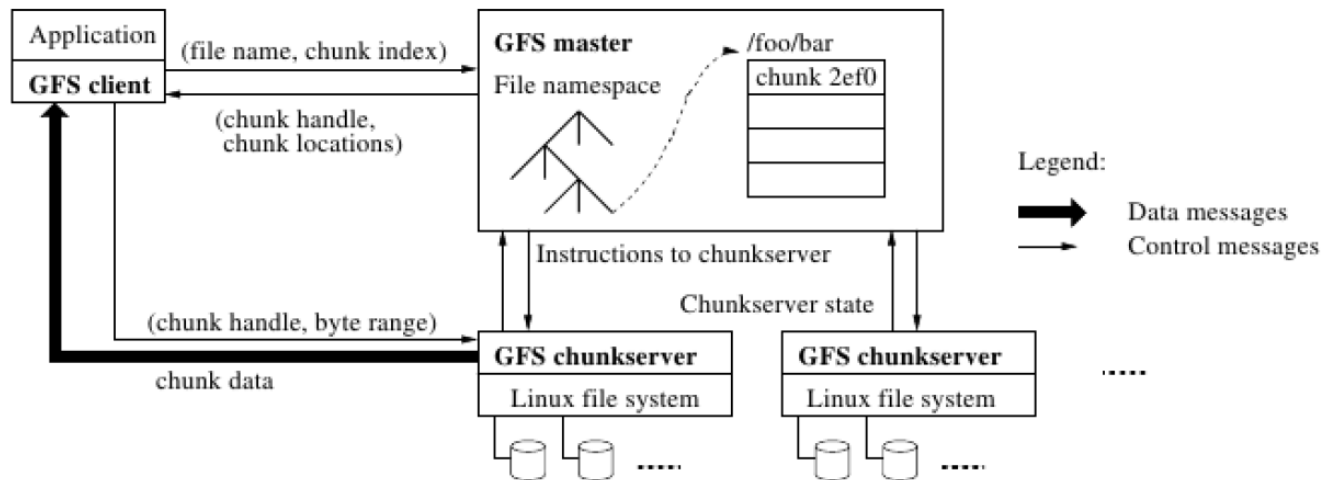


Fig 2.17: Architecture of a Google File System cluster [32.]

The second approach consists in distributing the meta information on several levels and therefore on several servers acting together as the central authority but having different roles. Splitting the meta data on several servers, without replicas and using a tree like structure allows distributing the load without having to use complex and costly replication protocols.

Note that the systems using this kind of architecture to store the meta data can afford to avoid storing replicas because they are running on top of other distributed file systems that handle the replication for them. Google's Bigtable. [34.], is an example of such a system with the root of its meta data architecture stored in the distributed lock service Chubby [35.] that stores the location of the servers serving the Root tablet, itself containing the location of the several METADATA tablets. To get a specific row stored in Bigtable, a new client has to connect to all the levels of the tree, but the information obtained on the upper levels are cached, meaning that further requests for data stored in tablets already looked for, will directly be made to the last level of the tree. The meta data architecture of Bigtable is shown on Figure below.

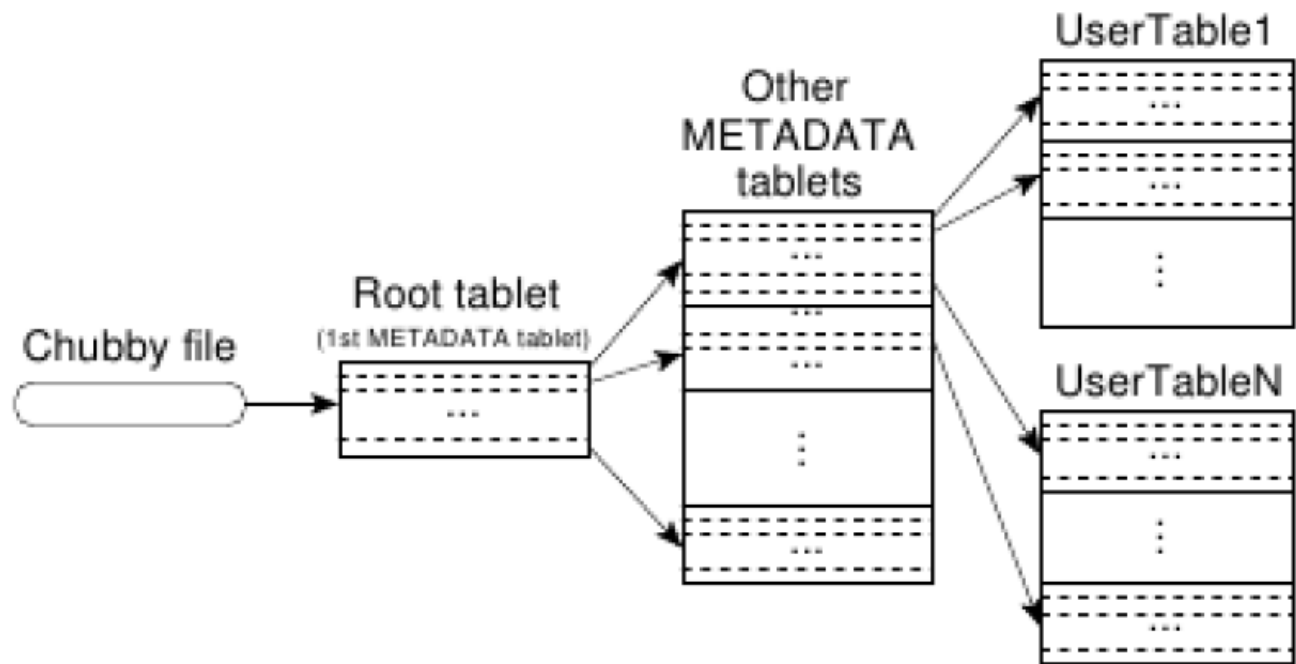


Fig 2.18: Architecture of the meta data in a Bigtable cluster

2.5. The CAP Theorem

The CAP theorem was first stated by Eric Brewer as a conjecture [36.] in 2000, it has been demonstrated in 2002 by Seth Gilbert and Nancy Lynch [37.].

The CAP theorem states that a distributed computer system can only provide simultaneously two of the following guaranties:

- Strong Consistency: this property is often stated as the atomicity of the objects stored in the database² and in practice it implies that every read that happens after a write operation completes will return the value written.
- Availability: it means that every requests sent to a working node in the cluster must result in a response.
- Partition tolerance: the nodes continue to work (they answer to read and write requests) even if the cluster is divided in subsets that cannot communicate together.

This theorem is criticized [38]. Because it looks like it would be possible to have every kind of combination, CP, AP and CA. But in practice, every system wants to be tolerant to partition. In fact the only question of interest regarding CAP is : what do we do when some nodes are unavailable? This is why the CA choice does not really make sense, it would be a system that does not tolerate network

partition, but what would happen if a partition occurs is that it would lose availability. Therefore, the choices provided by the CAP theorem are CP and AP.

Examples of real world approaches of the CAP theorem

As it relates to the CAP, the trade-off chosen by Google to implement Bigtable [34.] is partition tolerance and consistency. That implies that, if a node dies, some data will be unavailable until the remaining nodes start serving those data. This is due to the fact that there is no replication on the Bigtable level, rather the replication takes place one level below, at the Google file system level which stores all the data.

Amazon chooses availability and partition tolerance for Dynamo, meaning that it is still highly available even under network partition. Of course strong consistency is no more accessible and is replaced by eventual consistency. Inside a Dynamo cluster, the data is replicated at the Dynamo level itself. That implies that even if the network is partitioned, most of the data are still available. As the system is only eventually consistent, every reader is only guaranteed to get the last version of the data eventually.

2.6. Distribute the computation

Distributing the data across the cluster is a good start but it is still not enough to make computations on large subsets of the data. Indeed a way of using the computational power of all the servers and therefore a way of distributing the computation across all the nodes is needed. Moreover, it is relatively intuitive that moving the computation is cheaper than moving the data itself to nodes that would do the computation.

A solution to this problem was provided by Google in its Map Reduce paper [39.] and those concepts are now implemented in more and more distributed systems. It has the advantages of automatically distribute the computation across the node while taking into account the locality of the data, meaning that the work is done by the nodes that actually store the data.

Map Reduce concepts

Map Reduce is a programming model with its associated framework designed to process and generates very large set of data. It is based on Map and Reduce functions that the user must convert his computation into, generating a set of key/value pairs from another set of key/value pairs taken as input.

The Map function takes as input a key/value pair and produces a set of intermediary pairs. From those pairs, all the values associated to the same key are grouped together by the framework and those new pairs are sent as input of the Reduce function. The Reduce function takes as input an intermediate key and its associated set of values to merge those values into a possibly smaller set.

To explain how Map Reduce works in practice, here is an example of how to count the number of occurrences of each word in a set of documents with each document identified by a unique key and an associated value equal to the content of the document:

- The Map function is called once for each document and emits the pair (w,1) for each word w encountered
- The Reduce function sums up the all the values associated with each key and return the sum

Map Reduce implementations

The various implementations of Map Reduce do not differ much from the original regarding the basic concepts. Nevertheless, they differ in their purpose from one another.

The original idea behind Map Reduce was to create a framework that would easily parallelize heavy batch computations on large data sets distributed over a lot of servers. Those batch computations often requires to process lot of raw data to build things like inverted indexes, graphs or data analytic. Those computations need to be fast but the result is not needed immediately, therefore a high throughput is better than a low latency. This is the approach taken by Google and Hadoop.

After the release of Google's Map Reduce paper, others decided to use those concepts to build real time oriented query languages for distributed databases. Indeed a lot of those databases previously lacked query languages that could do more than the basic read and write operations. Therefore, they have implemented the concepts of Map Reduce in a very similar way, but they are not meant to be used on big set of raw data, instead they should be used on relatively small databases fields. This is the approach taken by Riak and mongoDB.

2.7. Study of some selected database

2.7.1. HBase

General presentation

HBase was derived from or is a clone of Google's Bigtable, it provides a real-time, distributed, versioned and structured database on top of the Hadoop distributed file system. Exactly like Bigtable, the system is made of two layers. First the distributed file system HDFS [40], [41], [42.] stores all the data in a persistent way, meaning that it is responsible for data replication, node failure and data distribution across the nodes. The second layer is the one made of HBase itself, where each region server is responsible for a list of regions meaning that it has to record the updates and writes into memtables and it also acts as a cache for the data stored in the HDFS level.

It is interesting to describe the architecture of an HBase cluster because it is much more complicated than a system where all the nodes are equal.

The architecture of a typical HBase cluster is shown in Fig 2.19

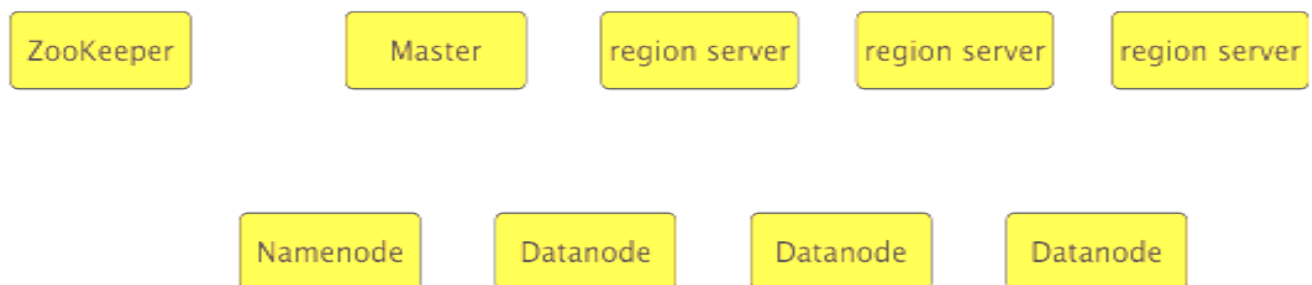


Fig 2.19: Architecture of a HBase cluster

A HBase cluster is in fact two distinct clusters working together, often on the same servers but not necessarily. The HDFS cluster is composed of one namenode that acts as the entry point of the cluster, meaning that this particular node knows which are the datanodes that store any information wanted by a client.

The HBase cluster is composed of one master, a Zookeeper cluster that acts as the entry point of the cluster and finally the region servers that serve the data. The masters and region servers are all registered into the Zookeeper quorum and if one of them dies it is repaired and replaced [43.] by Zookeeper. The master is doing background administrative tasks such as keeping the cluster balanced and moving regions from failing regions servers to other region servers.

The fact that both levels have a single point of entry can look like a bottleneck that would be overloaded very fast, but in fact it would not be the case in the short term. The [Figure 2.20](#) shows all the requests that must be made when a client connects to a newly bootstrapped HBase cluster and asks to read an entry in the database.

First, the client contacts one of the node of the Zookeeper quorum to know what is the address of the region server that stores the -ROOT- table.

Then the client makes a lookup into the -ROOT- table to find which region server is storing the .META. table to make a new lookup into this last table and find which is the region server that serves the region that contains the wanted entry.

Note that in the [Figure 2.20](#) it is assumed that the region server that stores the .META table also stores the wanted region and therefore those two requests have been merged for the sake of readability.

The region server contacted by the client has to do a similar work than the client to locate the data into the HDFS cluster before it can answer to the client, but with only one lookup. First it contacts the namenode to know the address of one of the datanode that is currently storing the data it is looking for and then contact directly the datanode to get the data.

Once a client is connected to Zookeeper, it can send as many requests as it wants, the connection will still be valid as long as there is activity at interval smaller than the configured timeout. That implies that, most of the time, the client does not have to recontact the Zookeeper quorum once a connection has been established. The client is also using cache mechanism to avoid making new

lookups in the `-ROOT-` and `.META.` table to find where is

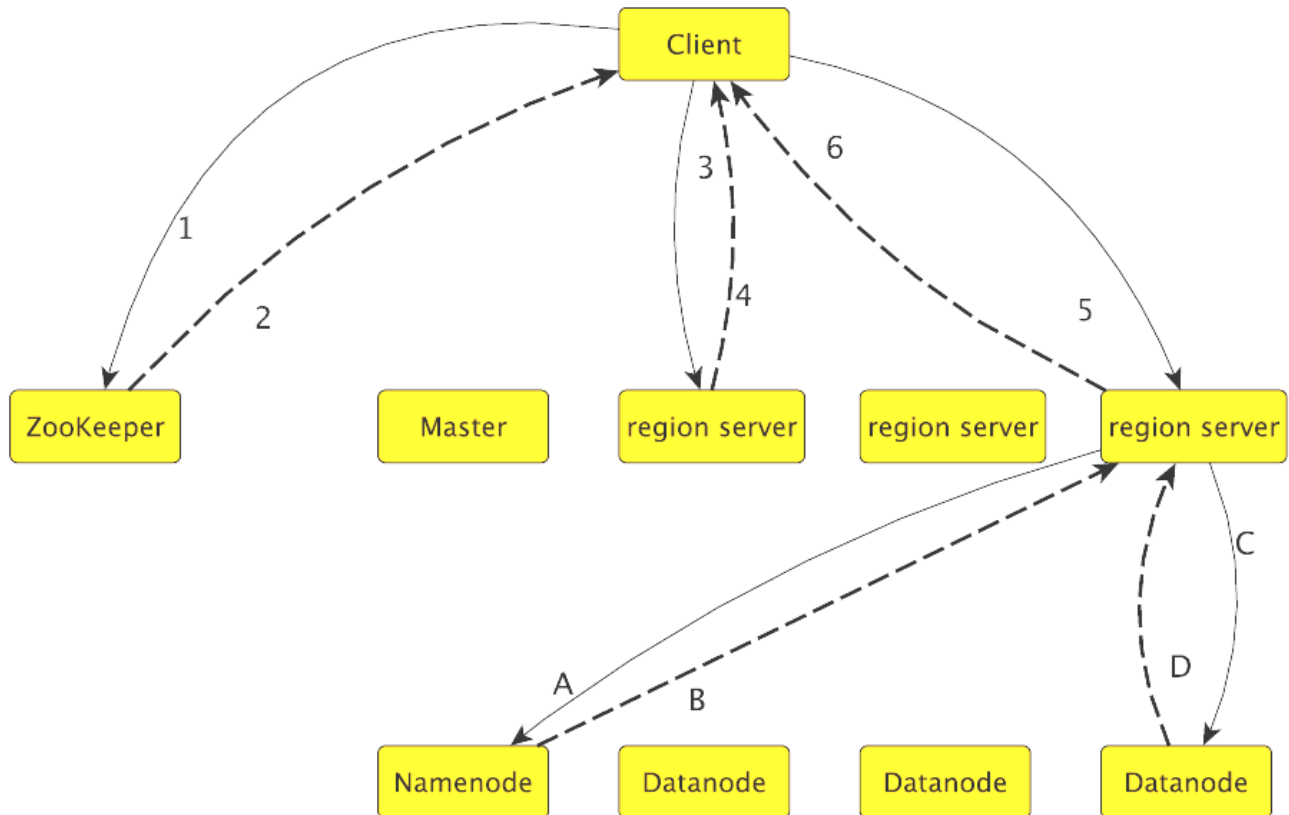


Fig 2.20: Requests made into a newly bootstrapped HBase cluster to get a database entry

a region that it already has asked for. A similar cache mechanism take place when the region server acts like a client to the HDFS cluster and therefore, if the data previously requested belongs to a block already seen, the client will not have to contact the namenode anymore. These cache mechanisms imply that after a while, most of the requests done into the cluster to get a database entry will look like the ones shown in [Figure below](#). Therefore, the centralized part of an HBase cluster would not be overloaded very fast and the bottleneck is pushed back thanks to the cache mechanisms.

Data model

The HBase data model can be said to be based on the following concepts [\[44.\]](#), [\[45.\]](#):

- The Tables: they are made of rows and columns

- Every Column belongs to a given column family
 - Each row is identified by its key
 - A Table cell is the intersection of the a row and a column and is versioned
- The Tables are created at the schema definition but new one can easily be added during normal operations. The number of versions is defined at the column family creation step. Note that HBase stores everything as uninterrupted bytes, meaning that almost everything, even a serialized data structure, can be used as name for a column family or as key for a row. The overall data model is represented in Figure showing HBase's data model representation.

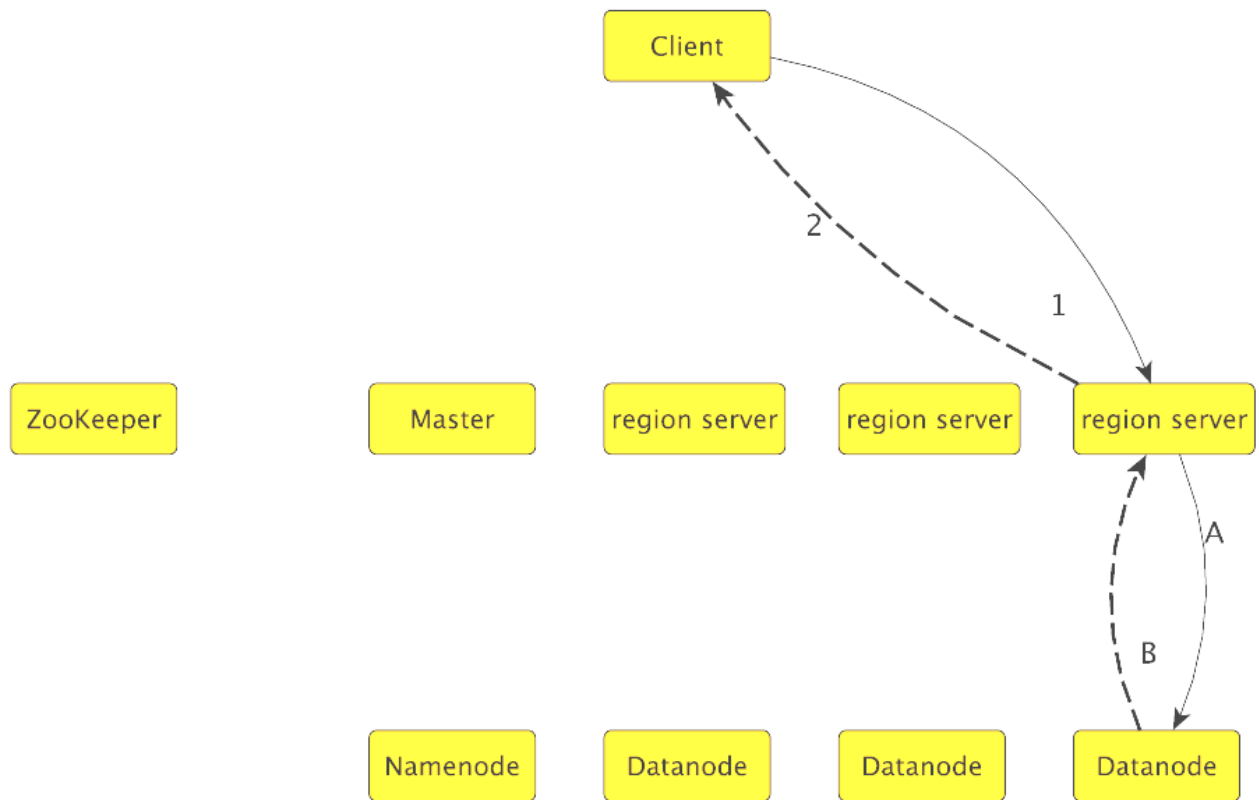


Fig 2.21: Requests made into an HBase cluster running in production to get a database entry

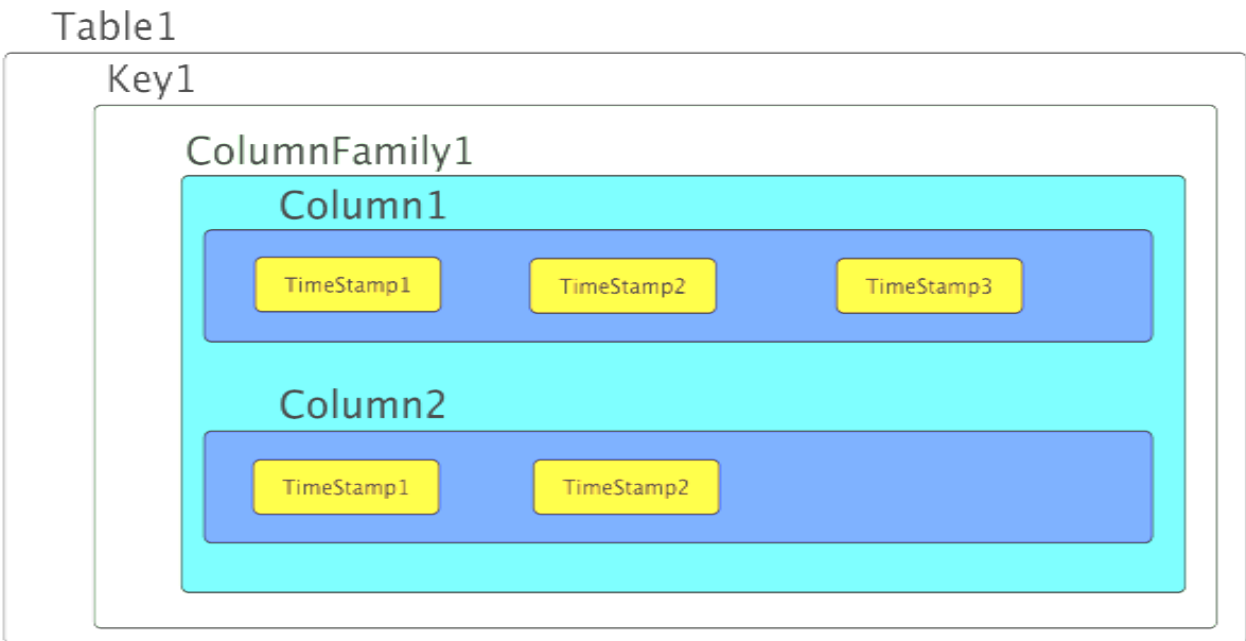


Fig 2.22: HBase's data model representation

In the same way Cassandra works, this data model is not static. New columns can always be added to existing column families even if only a small number of rows are actually using those new columns. This is due to the fact that the data are physically stored on a per column family basis, meaning that sparse data do not add any overhead into the stored file. However, as the optimizations are done on the column family level, it is recommended to store into the same column family columns that have similar access pattern and size.

The versions are not mandatory and the number of versions can be set to one at the column family creation step. If multiple versions are used, the client can either specify a given version or nothing to get the last version available.

Query model

With HBase, the user of the database can read data using either Gets or Scans. The Gets are used to retrieve all the columns or only a subset of them for a given row. The Scans are used to get all or part of the rows in a given Table. Like the Get operation, the subset of wanted columns can be specified but unlike it, it is also possible to define a range of rows using rows keys for the start and for the end of the range.

HBase Tables can be used as inputs and output for Hadoop MapReduce jobs when more complex computation than simple value lookup or update is needed. The

computation will be distributed [46] according to the number of regions available in the cluster, meaning that the number of Map jobs will always be equal to the number of regions.

CAP approach

For HBase, the selected CAP approach is CP meaning that if one node dies, the data that was served by this node will be unavailable for a moment.

This is due to the fact that the replication takes place at the HDFS level and not at the HBase level. The advantage, when only the HBase cluster is considered, is that if the server is up, all write and read operations served by this particular server will succeed. Indeed, as there are no replications at its level, a network partition between the region servers has no impact on the operations and every entry is always consistent as it is always served by a single server. But that implies that availability is sacrificed, indeed if a region server dies, all the data served by this region server will be unavailable until the Zookeeper quorum detects the failure and redistribute the existing regions across the remaining region servers.

HBase is not an ACID compliant database and does not provide any transactional support, but it provides certain number of guarantees [47.] :

- Atomicity: Every mutation⁹ is only atomic within a row even if the impacted columns belong to several column families.
- Consistency: Every single row returned is a complete row that existed in the Table history. However Scans are not a consistent view of a Table when all the rows returned are considered even if each of the row taken alone is consistent.
- Isolation: When a full row is retrieved concurrently with n mutations, it is guaranteed that the returned row will be a complete row that existed between two of the concurrent mutations.
- Durability: All data visible to the clients is stored durably. Therefore each operation returning success had been made durable and each operation returning failure has not.

Data replication and distribution

The replication takes place into the HDFS level but is configured inside HBase configuration files and can be set to any value greater than one. At both level of an HBase cluster, there is a single entry point that knows the chunk of data (i.e. blocks for HDFS and regions for HBase) that stores the wanted value. This entry point also knows the node (datanode for HDFS and region server for HBase) that serves this particular chunk of data. The namenode and the master do not only

know how the data is distributed across the nodes, they also choose how to distribute it. Therefore, the distribution of the data across the cluster is not a distributed process but is entirely handled by the entry point of each level. The HDFS's namenode distribute, almost evenly, newly written data to the existing datanodes in the cluster. But if new datanodes are added to the cluster, the existing data will not be moved, leading to an unbalanced cluster. To resolve this, new data is written to the cluster as in empty datanodes. It is also possible to call the balancer to manually balance the cluster by moving blocks between overloaded and under loaded datanodes. On the other side, the HBase's master will always try to allocate exactly the same number of regions to each region server available in the cluster. This process is entirely automatic and no human interaction is needed to balance the HBase cluster.

2.7.2. Cassandra

General presentation

Now let's consider the Cassandra database. The Cassandra [48.], [49.] is a fully distributed share nothing database. It is fault resistant, persistent and it can be used with a Hadoop cluster to run Map Reduce jobs. Cassandra tries to take the best from both Bigtable and Dynamo to build a new kind of distributed database. The features inspired by Dynamo are:

- Every node in the cluster is equal
- The cluster management is based on Gossip
- The consistency level is variable

On the other hand, the features inspired by Bigtable are :

- The column oriented model that allows efficient storage of sparse data
- The persistent storage format: SSTable [50.]
- The in memory storage format: Memtable [50]

Data model

The Cassandra data model [51.] is based on the following concepts:

- The Keyspace: a namespace for the ColumnFamilies
- The ColumnFamilies: are sets of Columns
- The SuperColumns: are Columns that can themselves contains Columns
- The Columns: are defined by a name, a value and a timestamp

Usually there is one Keyspace for each application that uses the Cassandra cluster. The Keyspace configuration is very important as it is the one that defines the replication factor as well as the replica placement strategy. The ColumnFamilies can be seen as table in traditional RDBMS; they are defined in static configuration files and cannot be changed during execution of the cluster.

New data is added in form of a row, like in a relational database, except that the column of the row can be different for each row. Each row is identified by a unique Key and it contains one or more ColumnFamily that contains Columns themselves. To help visualize, the [Figure below](#) represent those concepts.

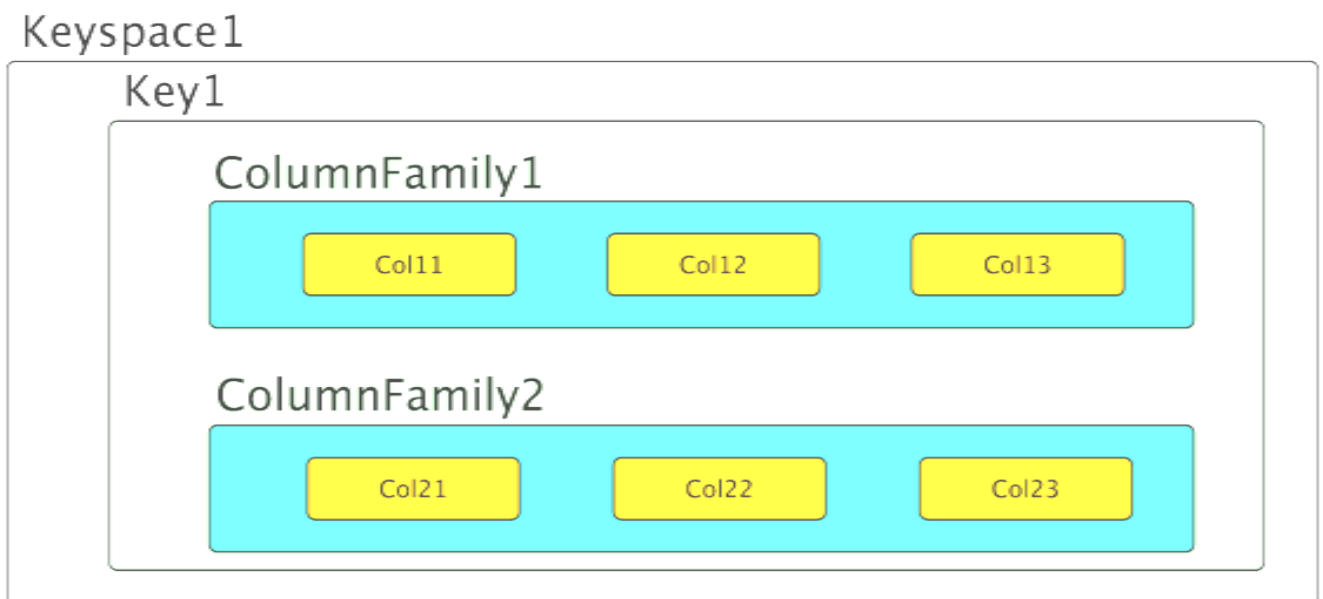


Fig 2.23: Cassandra data model : the two rows are identified by Key1 and Key2

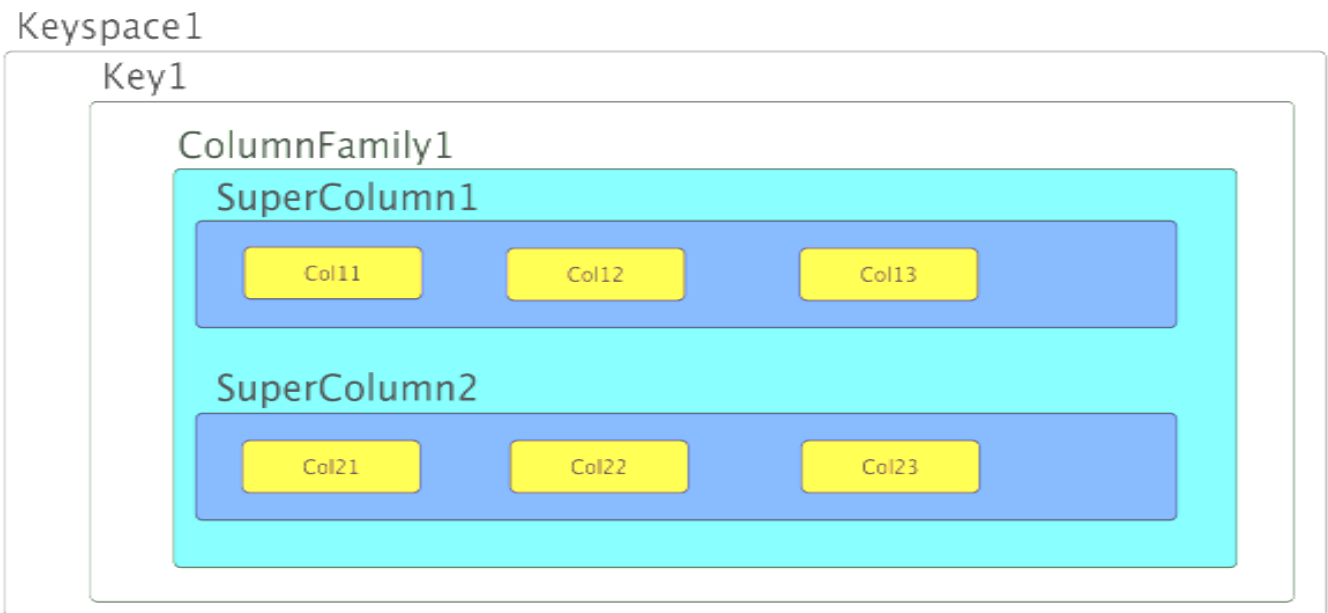


Fig 2.24: Cassandra data model: The ColumnFamily model extended to SuperColumn

In the example shown in Figure showing the two rows are identified by Key1 and Key2, two SuperColumn could have been used to group the Columns of the two ColumnFamily into a single SuperColumn but keeping a subdivision, adding therefore a level in the data structure.

This is illustrated in Figure showing The ColumnFamily model extended to SuperColumn.

Query model

The query model provided by Cassandra allows the user to access directly (For both read and write operations) to a particular column in a particular row as well as it is possible to get the whole line. It is also possible to define Slice Predicate, similar to mathematical predicate that is described as a property that the elements of a set have in common. [52.]

The Slice Predicate is useful when a subset of Columns is needed, it can be defined in two ways:

- With a list of columns names
- With a Slice Range describing how to range, order and/or limit the slice.

Finally, if Cassandra has access to a fully functional Hadoop cluster, Hadoop MapReduce jobs can be used to make complex computations on the data stored into Cassandra.

CAP approach

Coming to the issue of CAP, Cassandra has chosen availability and partition tolerance over consistency [53].

But, because Cassandra has chosen to let each request decide what the consistency level is, it is possible to achieve strong consistency by using a quorum method. The drawback compared to a system that has chosen consistency like Bigtable is that it is impossible to lock a row. Therefore one can be sure that one will always read the last write in a single Column but one cannot do the same for a whole row. On the other hand, the big advantage is that one never loses availability and therefore writes and reads should always succeed, depending on the consistency level chosen. For example with a replication factor of 3, if one node dies, read and write are still working as there are still a majority of nodes. However if a second node fails, quorum read and write will not work anymore while read and write using eventual consistency will be working.

With Cassandra, the developer can choose both the replication factor N , the number of nodes that must respond for a read operation R and the number of nodes that must respond for a write operation W . Cassandra guarantees [53] that strong consistency will be achieved on the impacted Column if and only if $W + R > N$, but as explained in Appendix A.1, this is not really strong consistency but time-line consistency.

Not really strong consistency but time-line consistency.

Data replication and distribution

With Cassandra it is possible to choose how the data will be partitioned and therefore change the way the data is distributed among the nodes. The first parameter of interest is the Token that is assigned to each node [14]. It will determine what the replicas that each node is responsible for are. The Tokens can always be strictly ordered, that allows the system to know that each node is responsible for the replicas falling in the range (Previous Token, Node Token]. The node assigned with the first Token is responsible for all the replicas from the beginning to this Token. Similarly, the node assigned with the last Token is responsible for all the replicas falling after this Token.

Those Tokens can be assigned automatically in a random way when the cluster bootstrap or they can be assigned manually.

To make the link between data and the corresponding Token, Cassandra uses partitioners. Out of the box, there are three partitioners available:

- The Random Partitioner implies that the Tokens must be integers in the range [0, 2127]. It uses MD5 hashing to compare Keys to the Tokens and therefore convert them to the range. If this partitioner is used, the Token selection is very important. Indeed, on average the Keys will be spread evenly across the Token space but if the chosen Tokens do not divide the range evenly, the cluster can be unbalanced⁴.
- The Order Preserving Partitioner will use the Keys themselves to place data on the ring, it is therefore the programmer duty to find good Tokens that ensure that the data will be evenly distributed across the nodes. If there is a “hot-spot in the ring, a new node can be added to the specific range. The advantage of using this partitioner is that range queries can be used. Note that this ordering is based on naïve byte ordering. The problem with this partitioner is that if the cluster must store several ColumnFamilies and if the distribution of the Keys is different, some nodes will end up storing much more Keys and the associated data [54.].
- The Collating Order Preserving Partitioner is similar to the Order Preserving Partitioner except that the ordering is based on EN and US rules to build a locale-aware ordering.

Cassandra also provides a way to customize the replicas placement strategy to take into account the cluster’s physical (rack and data center) topology.

By default Cassandra stores the replicas on the next $N - 1^5$ replicas along the ring in the same data center. If the Rack Aware Strategy is used, the second replicas will be stored along the ring but in another data center.

The $N - 2$ replicas left will be stored in the same ring but in the same rack than the first replica.

2.7.3. MongoDB

General presentation

MongoDB is a scalable database it is also schema free and document oriented [33.]. It is fault tolerant, persistent and provides a complex query language as well as an implementation of MapReduce.

The architecture of mongoDB can take several shapes depending on the needs.

The simplest case is the one where the database is not sharded but provides replication using slaves. This architecture is shown in [Figure below](#) and also shows a typical request sequence when a client wants to read or write a document. First the client directly asks to the master and gets its answer back.

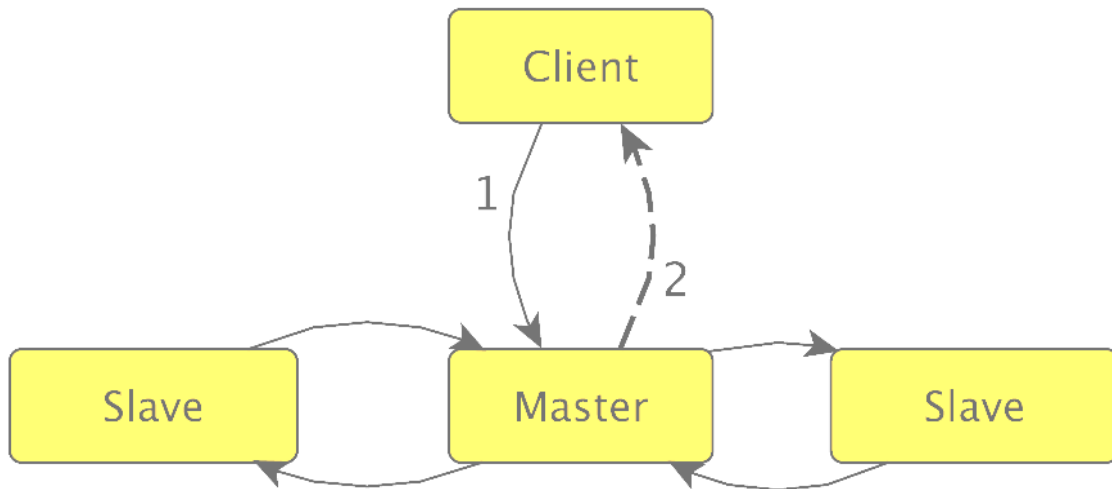


Fig 2.25: mongoDB not sharded with replication architecture

When the processing power or the storage space provided by a single master is not enough, it is possible to switch to a distributed architecture where the documents are divided between shards. Each shard can be a single mongod process or a replica set that is a set of servers running mongoDB. The servers belonging to the same replica set elect a master and all the other servers of the set are designated as slaves. The cluster also needs to know which the shards that currently store each document are. This information is stored into the configuration servers. Finally, the client is agnostic of the sharding, it talks to a mongos process that offers exactly the same API than an unsharded cluster. When a client wants a specific document, it asks for it to the mongos process that contacts one of the configuration server to know which is the shard storing the chunks that contains this documents and then contact directly the right shard's master to get the document and send it back to the client.

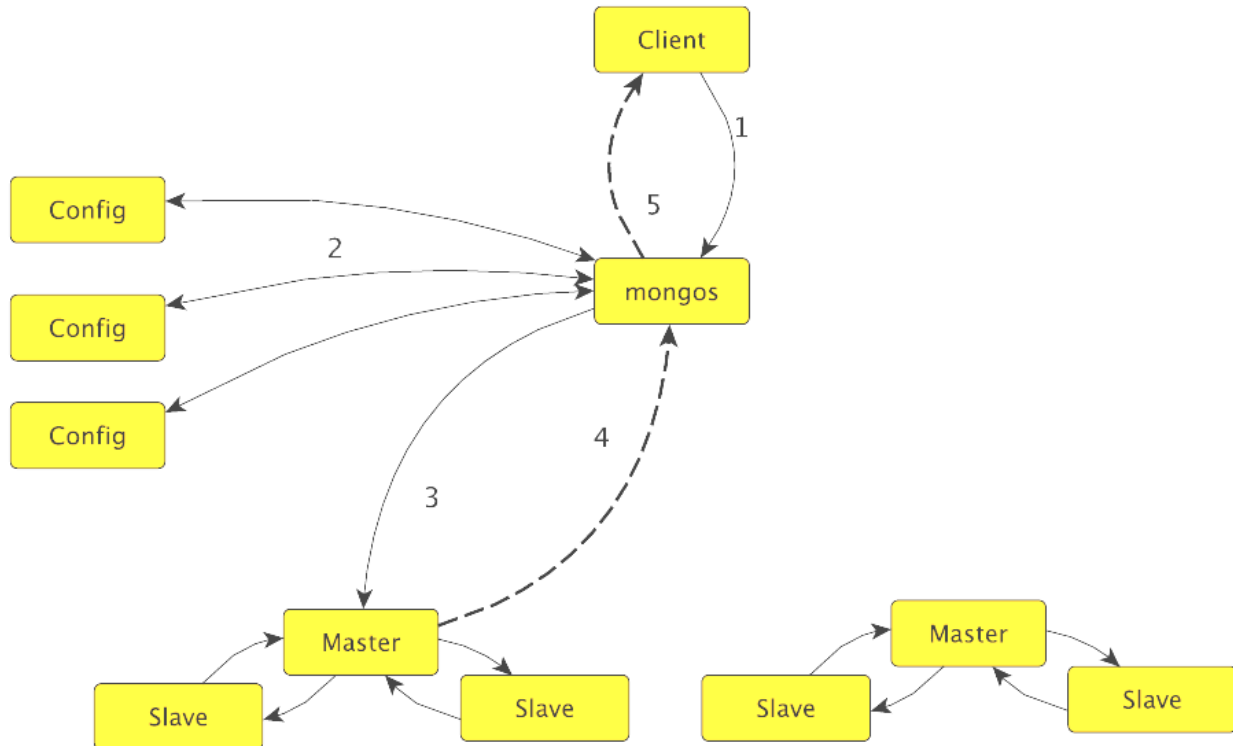


Fig 2.26: MongoDB sharded cluster with replication

Data model

The mongoDB data model [55.] is based on databases, collections and documents. The databases are sets of collections and collections are sets of documents. Each document is inserted as a JSON [56.], [57.] dictionary with a special field that can be provided at insert time or automatically generated by mongoDB. This field is named `_id` and is the default primary key of the document meaning that using the value of this field in a request allows for direct access to the document. Any other field can be added at any time during normal operations and no document is required to have other particular fields. This is why mongoDB is said to be schema less. The general data model of mongoDB is represented in [Figure below](#).

Database1

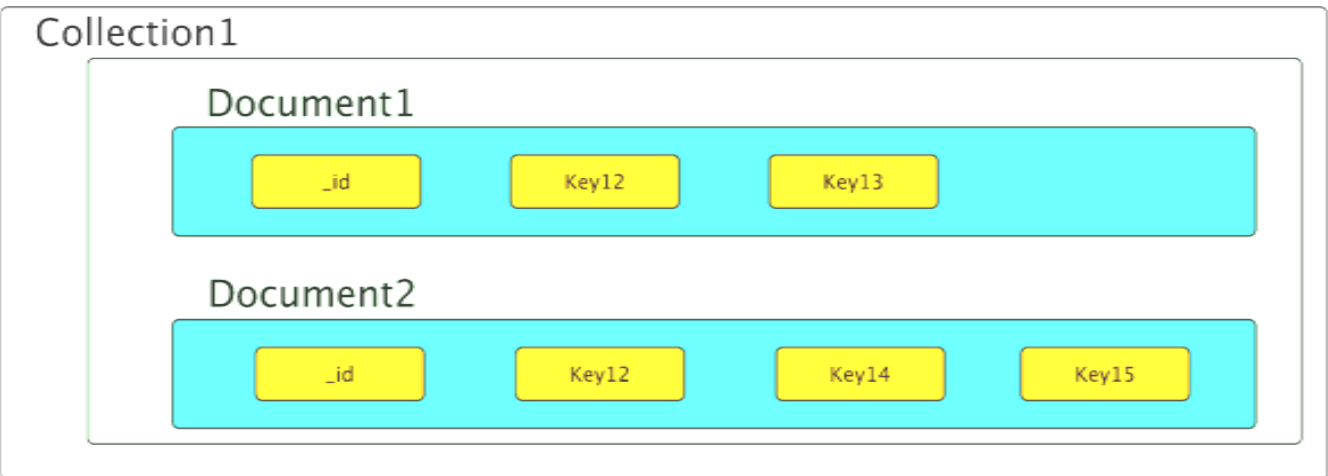


Fig 2.27: mongoDB's data model representation

Query model

The query model of mongoDB is rich compared to other NoSQL databases.

In fact it is closest in its expressiveness, with the notable lack of JOINS, to the SQL language.

Given the important number of possible kinds of queries with mongoDB, only an overview of its query environment is provided. The main functionalities provided [58.] by mongoDB are:

- Selection of a subset of documents using:
 - Exact field(s) matches
 - <_ ; >_ operators on single or multiple fields value
- The all operator, used to provide a list of items that must be present in the returned set of document
- The in operator, used to provide a list of items of which at least one item must be present in the documents returned
- The exists operator, used to select the documents that contain or not the selected field
 - Regular expression applied on the content of a chosen field
- The cursor methods used with other selection methods:
 - Count returns the number of document matching the query
 - Limit used to set the maximum number of results that can be returned
 - Skip used to skip the first n results
 - Sort used to order the documents returned using a given field
 - Distinct used to be sure that all returned documents have a unique value for the selected fields.

– Group used in a similar way as the SQL statement

In addition to those features, mongoDB provide a MapReduce implementation [59.]. The Map and Reduce functions must be written in JavaScript and are executed by a JavaScript virtual machine on the server side. It is interesting to note that currently this implementation of MapReduce is only single threaded due to limitations in the JavaScript engine. That implies that the only way of getting MapReduce jobs done in parallel is to execute them in a sharded environment where each shard will execute the Map function on the documents it stores.

2.9.3. CAP Approach

MongoDB can provide various levels of consistency, depending on then architecture used and the configuration of the clients. By default mongoDB is in the CP camp, meaning that all the requests are strongly consistent [60.]. With mongoDB, there is always a single master for the whole database or a single master for each shard in the cluster and by default all the read and write requests are sent to master(s). This is why all the requests are consistent by default. To enable a weaker form of consistency, it is possible to read from slaves. This will result in eventual consistency, meaning that if a request to read a document that has just been updated on the master is sent to a slave, it may read stale data.

As already explained before, with mongoDB the replication depends on the number of slaves running in each shard. It is therefore possible to have various level of replication from shard to shard.

To distribute data across the shards in the cluster, mongoDB relies on ranges that split the data in chunks of configurable size [61.]. The ranges are based on a sharding key that can be any field or a combination of fields. For a previously unsharded collection, mongoDB will create a single chunk whose range would be $(-\infty, \infty)$ with $-\infty$, and ∞ , the smallest and biggest values mongoDB can represent for the chosen sharding key. Then, until all the chunk sizes are at most equal to the configured chunk's size, mongoDB will split each chunk in half regarding to the sharding key. Existing chunks are also split using the same mechanism when their size reaches the configured chunk's size. Note that once a collection is sharded, it is not possible anymore to insert into it a document lacking the field(s) chosen as sharding key.

The sharding key has several important properties:

- There must exist a strict ordering between all the values used as sharding keys.

- Its cardinality will define the number of available chunks, therefore a high cardinality sharding key is recommended. Indeed when the number of chunks equals the number of available sharding key, it is not possible anymore to split chunks and therefore distribute new data.
- The sharding key must be chosen related to the application using mongoDB to avoid hot-spots. For example an application in which the newest content is the one generating the most traffic should not use an increasing ID as sharding key. Indeed the newest content would always be stored on a few chunks stored most of the time on the same shard.

The chunks are constantly and automatically balanced across the available shards by a process called the balancer. This process simply looks at the number of chunks on each shard and start moving chunks when the cluster is unbalanced. It is interesting to note that there is only one balancer process cluster wide, that implies that there will never be more than one chunk moved at a time.

As briefly explained before, the configuration servers are storing the cluster metadata. They know which shard is storing each chunk as well as basic information about each shard like the number of chunks they store. The cluster formed by the, usually three, configuration servers uses its own replication model, with each server storing all the metadata. Every change in those information is made using a two-phases commit to ensure that the data stored by this cluster are always consistent. Finally, this cluster becomes unwritable as soon as one of the configuration servers is down, preventing any other chunk split or move. This is not as bad as it may look because even if the cluster of configuration servers is read-only, all the read and write operations done on documents can still be processed by the cluster of shards but it will become unbalanced if writes are still sent to it.

Finally, the last players of a mongoDB cluster are the mongos. They are router processes that hide all the complexity of the cluster to the client who simply talk to a mongos as if it was a non distributed mongoDB installation.

It is considered as a routing process because it finds where the client's request must be sent using information stored in the cluster of configuration servers. Usually, there is one mongos process running on each application server. It is not recommended to use a single or small number of mongos processes because they act as proxies for the client and therefore could overload their own bandwidth very fast.

2.7.4. Riak

General presentation

Riak [62.] is a fully distributed shared nothing database, it is fault tolerant, persistent and provides an implementation of MapReduce.

Data model

The data are stored in Riak as key/value pairs and each pair is stored into a bucket. Each key in a bucket must be unique. The buckets can be seen as collections of key/value pairs, they can be used to apply different settings to sets of key/value pairs like different replication factors. Finally Riak also provide links that enable key/value pairs to be linked to one another. Those concepts are shown in [Figure below](#).

Bucket

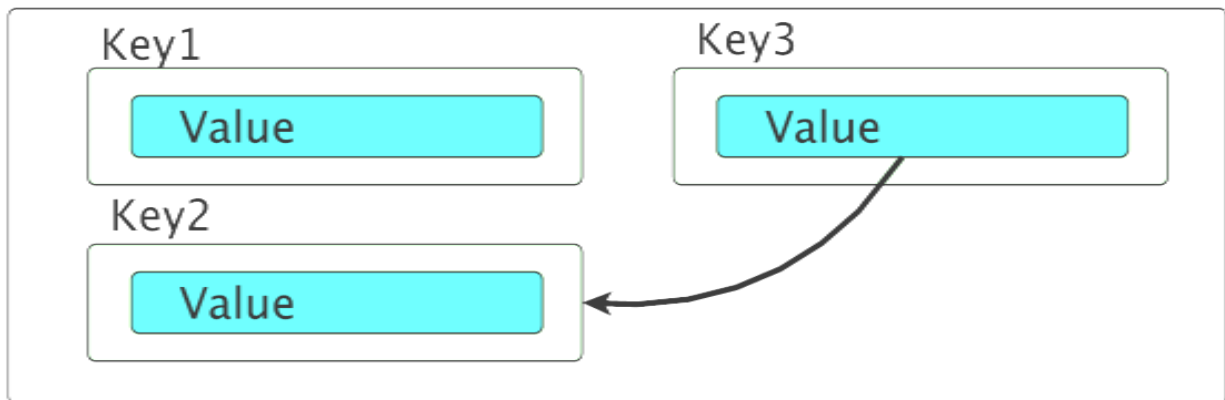


Fig 2.28: Riak's data model representation

Query model

The query model of Riak provide the usual Put, Get and Delete operations of the key/value stores to do standard read, write and update operations.

Every query that needs more complexity must be written as a MapReduce job. A MapReduce job is made of a series of phases that can be of three kinds:

- Map phases: can operate on entire buckets or lists of bucket/key pairs and must return a list of key/value pairs. Note that the keys returned mustn't be necessarily present in buckets.
- Link phases: are specialized versions of map phases that fetch objects based on a link walk. They can be used to perform MapReduce computations on related sets of objects.

- Reduce phases: can perform any kind of computation on results returned by a map phase or another reduce phase. It does not have the obligation to return a single answer.

On the technical side, communications with the Riak cluster can be done using the following interfaces: HTTP, Erlang, Python, Java and Ruby. All the MapReduce jobs must be written in JavaScript or in Erlang.

CAP approach

Riak is clearly inspired by Dynamo and choose A and P over C [63.] in the CAP theorem, meaning that if the network is partitioned or if there is a race condition in which two client update the same value at the same time on two different nodes, Riak will accept the write on both nodes. That implies that there will be a conflict when the cluster is whole again or very fast if it was due to the race condition. When a conflict appears, Riak let the client choose between getting the last updated version or to resolve the conflict by human-assisted or automated action.

It is possible to achieve time-line consistency with Riak using a quorum mechanism. The replication factor (n value) can be specified for each bucket and it is also possible to specify at each read or write request the number of nodes on which the operations should be done (respectively r and w). That implies that time-line consistency will be achieved if $(r + w) > n_value$.

This tunable consistency level allows the programmer to obtain greater performances when a lower level of consistency is needed.

Data replication and distribution

Each Riak cluster has a 160-bit integer space that is divided into partitions of equal size. The number of partitions must be fixed at cluster initialization.

Riak starts a virtual node for each partition in the cluster and then distribute evenly those virtual nodes across the physical nodes that have joined the cluster.

Riak uses consistent hashing to distribute data across the cluster. Each replica of a given object is stored on a different partition and Riak will automatically balance the cluster when new nodes are added or removed.

Finally, Riak is based on a share nothing architecture in which all the nodes are equal. It allows each node to be fully capable of serving any kind of request, the node can always find which are the nodes that store the information by using the consistent hashing.

2.7.5. Scalaris

General presentation

Scalaris data store is a distributed, transactional, non-persistent key/value store. This data store has been designed to be a self-managing and scalable distributed data store, meaning that it is possible to add or remove servers without any downtime.

Scalaris is a fully distributed data store with all nodes being equal, it takes care of:

- The failure of node using fail-over
- The distribution of the data across nodes
- The replication of the data
- The strong consistency of all the requests
- The transactions

To ensure that all those services are provided, Scalaris is made of three layers [64.] implemented in Erlang. Those layers can be seen on [Figure below](#)

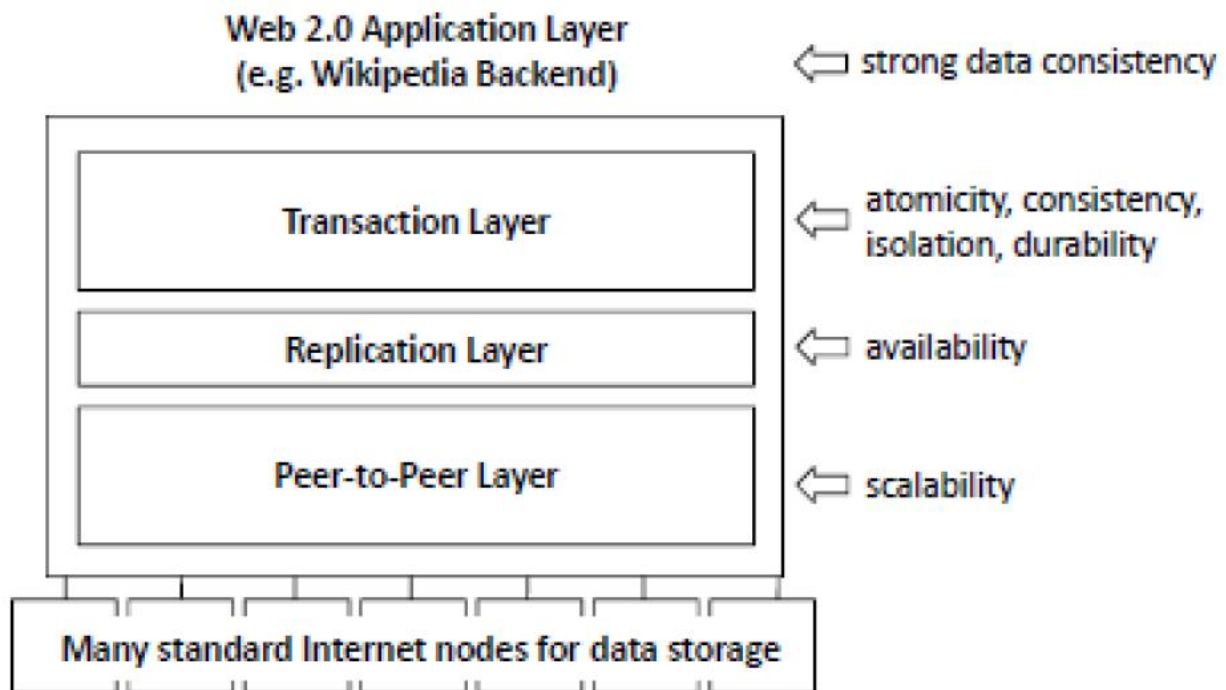


Fig 2.28: Scalaris architecture

Going from bottom to top, the layers are:

1. The peer-to-peer layer is a distributed hash table with logarithmic routing performances. It is the basis of the key/value store.

2. The second layer takes care of the replication of the data using symmetric replication.
3. The third layer implements the ACID properties and transactions using an improved Paxos protocol.

Data model

The data model provide by Scalaris [65.] is quite simple as it is only a key/value store without any other layer to structure the data. All the key/value pairs are stored into the same set of entries.

Query model

Given the simplicity of the data model, the query model is also very simple and is limited to the expected write, read and delete operations. Of course the big advantage of Scalaris is the possibility to use transactions and, as one could expect, they simply consist in grouping together normal operations into one transaction.

CAP approach

As briefly mentioned before, Scalaris is strongly consistent and supports transactions. Therefore, as the partition tolerance is something that is almost always wanted, Scalaris is in the CP camp [66.].

Of course that implies that Scalaris is willing to lose availability to make sure that the data will always be consistent. For example, in the case of network partition with two partitions for a cluster using a replication factor of 4, if there are 3 replicas in one partition and 1 replica in the other, the read and writes will only be available in the partition containing the 3 replicas. While all the requests sent to the other partition will all result in a timeout until the two partition are merged again.

Data replication and distribution

Scalaris uses consistent hashing to place each key on the ring [67.] and then the replicas are stored at 90° intervals after the first one. The use of consistent hashing implies that the position of a given key on the ring is random, and therefore so are the replicas.

The placement of the nodes on the ring is random but the developers have added a passive load balancer that is now activated by default¹⁶. This balancer ensures that the address range is uniformly distributed during a node join. For a cluster running for some time, meaning that the nodes have been in contact for a while and have aggregated the average load of the whole cluster, it will also split the actual load instead of the address space.

2.7.6. Voldemort

General presentation

Voldemort [68.] is a distributed, persistent, fault tolerant key/value store. It is fully distributed, meaning that each node is independent with no central point of failure or coordination.

Voldemort has been designed to be used as a simple storage that can be fast enough to avoid using a caching layer on top of it. The software architecture is made of several layers [69.], each of them implementing the put, get and delete operations. Each layer is responsible for a specific function like TCP/IP communications, routing or conflict resolution. Those layers are shown on [Figure below](#).

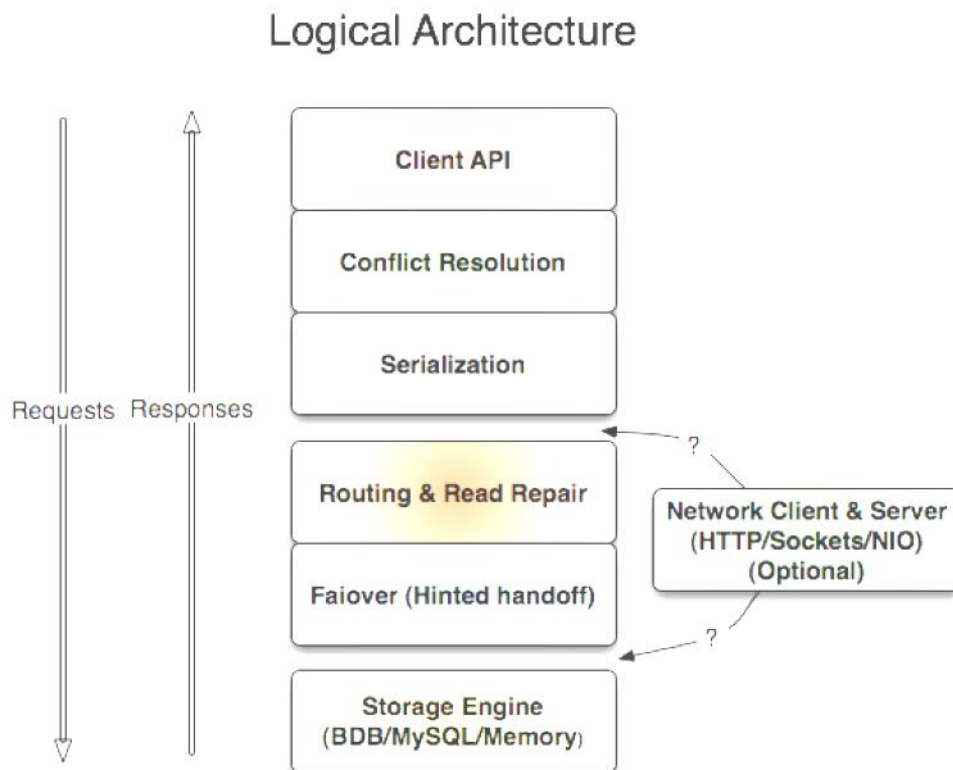


Fig 2.29: Voldemort architecture, taken from [23.]

This architecture allows those layers to be used in different combinations to meet different needs and also ease the addition of a functionality that would impact the whole system without any other modifications than the addition of a new layer. For example, if a fast and effective compression is needed, adding a compression layer just below the serialization one would do the trick.

Data model

The data model provided by Voldemort is a basic key/value store with the possibility to separate the keys in several stores that can be seen as tables. Therefore there is only one level of subdivision of the data inside Voldemort. However, as Voldemort supports various popular serialization frameworks, it can be used to store complex objects like maps and lists and therefore store relations between entries in the database. Of course that implies that all the logic is implemented at the application level.

Query model

The query model offers the usual get, put and delete operations once a store has been selected as the subset of keys to use.

CAP approach

Voldemort allows temporary inconsistencies and resolve them at read time to ensure the best availability possible. That implies that Voldemort is in the AP camp.

Voldemort uses read-repair and versioning based on vector-clock, meaning that is possible that some inconsistencies require additional application logic to be resolved. This is why there is a layer called conflict resolution in the architecture.

The replication factor N , as well as the number of required reads R and the number of required writes W are specified on a per store basis. If

$W + R > N$, Voldemort guarantees that a read issued just after a completed write will return the updated value. However the put and delete operations are neither immediately consistent nor isolated. Voldemort guarantees only that if a put or delete operation succeeds without returning an exception, it implies that at least W servers had carried the operation. If an exception is returned, the state of the data is unspecified and the client must issue a new write to ensure that the data will be in a consistent state.

Data replication and distribution

The key range is represented as a ring on which the keys are placed using consistent hashing. The ring is divided in Q partitions following the chosen value for this parameter and then each of the S servers is assigned $Q=S$ partitions. Once the position of a key on the ring has been computed using the hash function, the R replicas are stored on the first R unique servers encountered moving clockwise over the partitions.

Currently all the routing is done on the client side, meaning that the client has first to connect to a node in the cluster to learn the network topology. Once it knows which nodes is storing each partition, it can use the hash code that it computed itself with the wanted key to find which are the nodes storing this key.

Existing benchmarks

TPC

The TPC benchmarks [70.] are the most-used benchmarking tools in the RDBMS world. They are proven and mature ways of assessing the performance of transactional databases. The fact that those benchmarks test transactional performance is appropriate for a standard RDBMS environment but is very problematic for the databases concerned in this study.

Indeed none of the chosen databases support transactions. At best HBase supports row locking and the others only provide strong consistency on single fields. It is therefore not possible to use this kind of benchmark on those databases while enforcing the same consistency properties.

2.8. YCSB

The Yahoo! Cloud Servicing Benchmark [71.], [72.] is the most well-known benchmarking framework for NoSQL databases. It was created by Yahoo! It currently supports many different databases and it can be extended to use various kinds of workloads. The benchmark used for the measurements presented here, both concerning the storage and MapReduce, could have been implemented on top of YCSB as a new workload but it has not been for various reasons. The first reason is for simplicity: it seemed easier to implement its functionalities directly instead of extending the big and far more complex YCSB where it would not have been so easy to control all the parameters.

The second reason is that I wanted to explore the best methodology for measuring elasticity without being tied to the assumptions of an existing tool.

2.9. Catalyst 2950 Switch

Cisco Catalyst 2950 Switch was used for networking the systems because they are much better than ordinary hubs and they have more functions and easy to use.

(73)

A **network switch** or **switching hub** is a computer networking device that connects network segments or network devices. The term commonly refers to a multi-port network bridge that processes and routes data at the data link layer (layer 2) of the OSI model. Switches that additionally process data at the network layer (layer 3) and above are often referred to as *layer-3 switches* or multilayer switches. [85]

2.10. Matlab Description

MATLAB is a high-level language and interactive environment for numerical computation, visualization, and programming. Using MATLAB, you can analyze data, develop algorithms, and create models and applications.

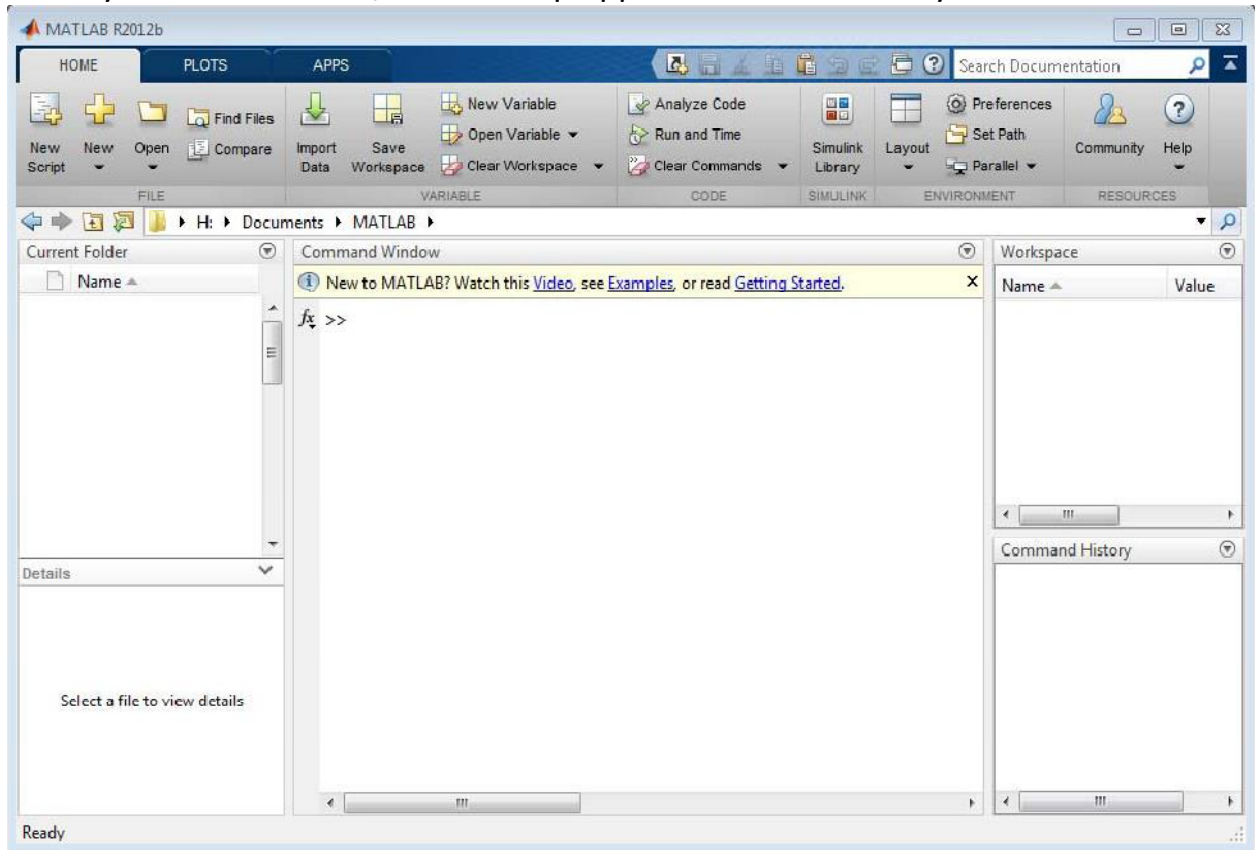
The language, tools, and built-in math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java™. You can use MATLAB for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology.

Key Features

- High-level language for numerical computation, visualization, and application development
- Interactive environment for iterative exploration, design, and problem solving
- Mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving ordinary differential equations
- Built-in graphics for visualizing data and tools for creating custom plots
- Development tools for improving code quality and maintainability and maximizing performance
- Tools for building applications with custom graphical interfaces

- Functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET, and Microsoft Excel.

When you start MATLAB, the desktop appears in its default layout.



Matlab Screen From [75] fig 2.30

The desktop includes these panels:

- **Current Folder** — Access your files.
- **Command Window** — Enter commands at the command line, indicated by the prompt (`>>`).
- **Workspace** — Explore data that you create or import from files.
- **Command History** — View or rerun commands that you entered at the command line.

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named `a` by typing this statement at the command line:

```
a = 1
```

MATLAB adds variable `a` to the workspace and displays the result in the Command Window.

```
a = 1
```

Create a few more variables.

```
b = 2
```

```
b = 2
```

```
c = a + b
```

```
c = 3
```

```
d = cos(a)
```

```
d = 0.5403
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of your calculation. `sin(a)`

```
ans = 0.8415
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

```
e = a*b;
```

You can recall previous commands by pressing the up- and down-arrow keys, $\uparrow$ and $\downarrow$. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command `b = 2`, type `b`, and then press the up-arrow key.

Array Creation

To create an array with four elements in a single row, separate the elements with either a comma (,) or a space.

```
a = [1 2 3 4] returns
```

```
a = 1 2 3 4
```

This type of array is a *row vector*.

To create a matrix that has multiple rows, separate the rows with semicolons.

```
a = [1 2 3; 4 5 6; 7 8 10]
```

```
a = 1 2 3
```

```
4 5 6 7 8 10
```

Another way to create a matrix is to use a function, such as `ones`, `zeros`, or `rand`. For example, create a 5-by-1 column vector of zeros.

```
z = zeros (5,1)
```

```
z =
```

```
0
```

```
0
```

0
0
0

2.11. Matrix and Array Operations

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

$a + 10$

Answer =

```
11 12 13
14 15 16
17 18 20
```

Sin (a)

Answer =

```
0.8415 0.9093 0.1411
-0.7568 -0.9589 -0.2794
0.6570 0.9894 -0.5440
```

To transpose a matrix, use a single quote ('):

a'

Answer =

```
1 4 7
2 5 8
3 6 10
```

Concatenation

Concatenation is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets `[]` is the concatenation operator.

$A = [a,a]$

A =

```
1 2 3 1 2 3
4 5 6 4 5 6
7 8 10 7 8 10
```

Concatenating arrays next to one another using commas is called *horizontal* concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate *vertically* using semicolons.

$A = [a; a]$

A =

```

1 2 3
4 5 6
7 8 10
1 2 3
4 5 6
7 8 10

```

2-D and 3-D Plots

In this section...

“Line Plots” on page 1-18

“3-D Plots” on page 1-21

“Subplots” on page 1-22

Line Plots

To create two-dimensional line plots, use the plot function. For example, plot the value of the sine function from 0 to 2π :

```
x = 0:pi/100:2*pi;
```

```
y = sin(x);
```

```
plot (x,y)
```

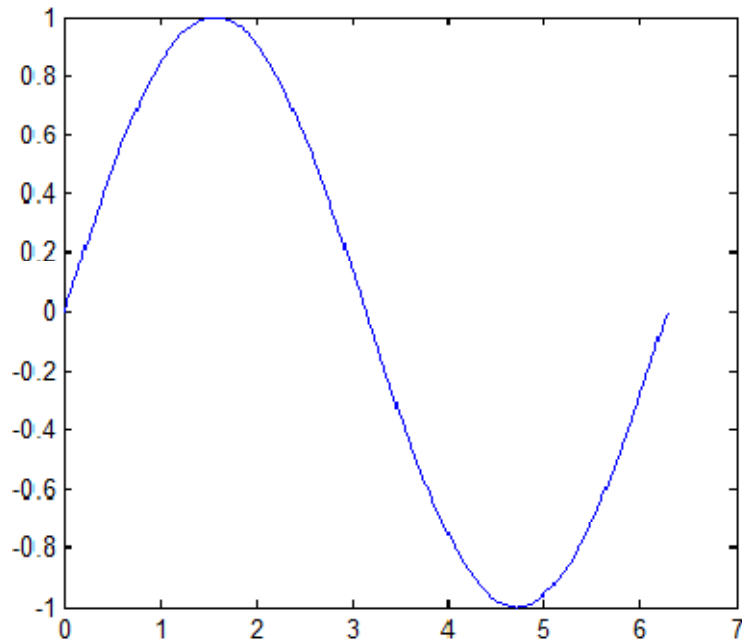


Fig 2.31 Line Plots

You can label the axes and add a title.

```
xlabel('x')
```

```
ylabel('sin(x)')
```

```
title('Plot of the Sine Function')
```

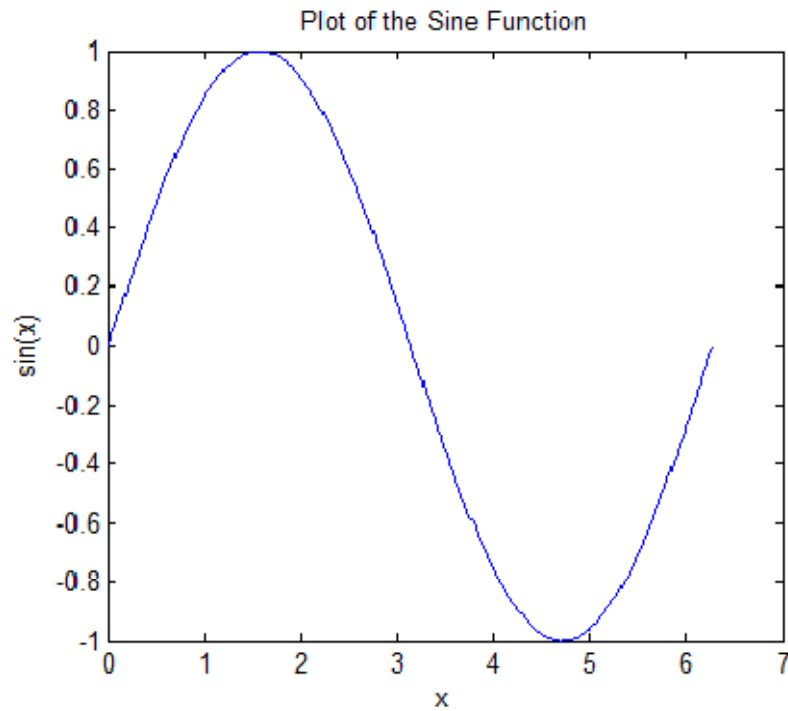


Fig 2.32 Sine function

By adding a third input argument to the plot function, you can plot the same variables using a red dashed line.

```
plot(x,y,'r--')
```

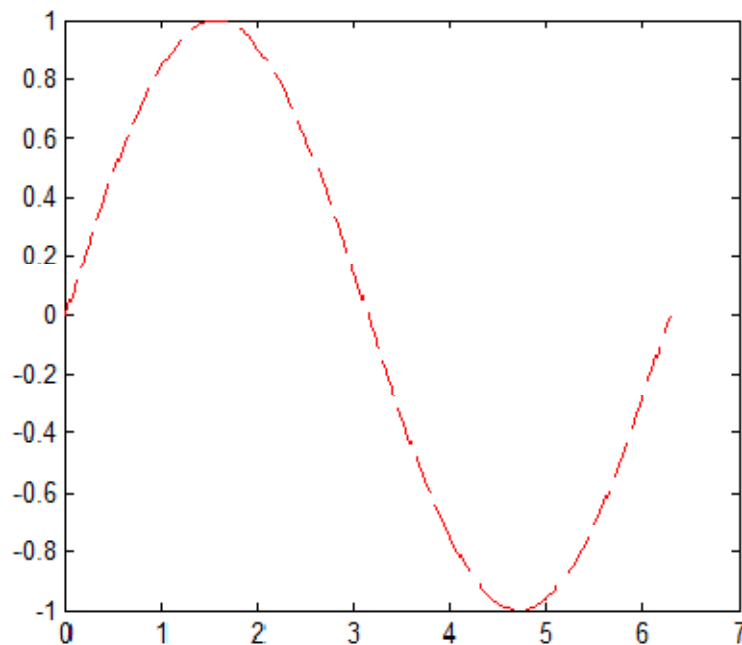


Fig 2.33 *line specification*

The 'r--' string is a *line specification*. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a +, o, or *. For example, 'g:*' requests a dotted green line with * markers.

Notice that the titles and labels that you defined for the first plot are no longer in the current *figure* window. By default, MATLAB clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot.

To add plots to an existing figure, use hold.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
hold on
y2 = cos(x);
plot(x,y2,'r:')
legend('sin','cos')
```

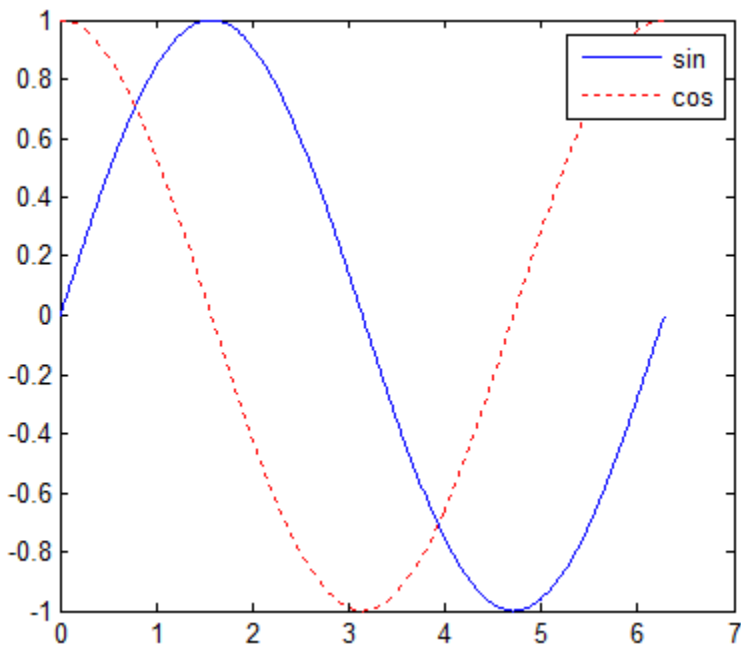


Fig 2.34 Cosine and Sin

CHAPTER THREE

METHODOLOGY

3.1. Parameters/Properties being measured

Three fundamental measures for distributed databases designed for “Bigdata” problems are performance, scalability, and stability. First, here are the definitions of these measures.

3.1.1. Performance

Performance can be said to be the time needed to complete a given number of requests with a given level of parallelization. The chosen levels of parallelization

and number of requests used during the measurements are explained in the step by step methodology. In all the measurements presented, I perform requests in batches called request sets. This allows us to decrease variability and improve accuracy in measurement time.

3.1.2. Scalability

Scalability can be defined as the change in performance when nodes are added and fully integrated in the cluster. In practice the size of the data set, the level of parallelization, and the number of requests are all increased in the same proportion as the number of nodes. Therefore, a system is perfectly scalable if the time needed on average to execute a request set stays constant when all the parameters of the cluster grow linearly. Of course this measure is only valid for clusters that are stabilized, meaning that all new and existing nodes have finished their partition transfers and are fully operational.

3.1.3. Stability

Stability can be said to be the time it takes the cluster to return to its normal data processing time after new nodes has been added.

3.2. Materials used

1. Switches (Catalyst 2950 Cisco Switches). 3 switches were used and they have 24 LAN ports each.

2. Systems used (System specifications).

I used a total of 24 systems for the comparison of different NoSQL models.

Name of Systems – Lenovo

Operating System – Windows Vista

Processor Speed – 4.0 GHz

RAM Size – 2G RAM

Hard Disk – 120G

3.2.1. Database Software configuration settings

Each database needs some specific configuration settings, especially concerning the memory and the way data is shard across nodes. This section also explains how data is structured in each database.

3.2.2. Cassandra

For Cassandra, only one Keyspace with three Columnar Families was created. Cassandra shards data across nodes following two parameters, the node's Token and the selected Practitioner. The Tokens will determine what the replicas that each node is responsible for are. The Tokens can always be strictly ordered, which allows the system to know that each node is responsible for the replicas falling in the range (PreviousToken, NodeToken]. The node assigned with the first Token is responsible for all the replicas from the beginning to this Token. Similarly, the node assigned with the last Token is responsible for all the replicas falling after this Token. Those Tokens can be assigned automatically in a random way when the cluster bootstrap or they can be assigned manually.

To make the link between data and the corresponding Token, Cassandra uses partitioners. Out of the box, there are three partitioners available.

We chose the Random practitioner. This implies that the Tokens must be integers in the range $[0, 2^{127} - 1]$. It uses MD5 hashing to compare Keys to the Tokens and therefore convert them to the range. If this partitioner is used, the Token selection is very important. Indeed, on average the Keys will be spread evenly across the Token space but if the chosen Tokens do not divide the range evenly, the cluster can be unbalanced. To solve this problem I generated the Tokens with:

$$\text{Token}_i = i * 2^{127} / N$$

for $i = 0 \dots N - 1$ with N the maximal number of nodes in the cluster. In our case, with the given budget, I generated tokens for 24 nodes, which was the most nodes that I projected to use. Once the tokens are generated, the first 6 nodes had for Token the tokens that divided 24 in 4 equal parts. Then each time the cluster's size is doubled, the node tokens are the ones falling exactly in the middle

of the token already used. This way, at each point, the percentage of data stored on each node is exactly the same.

3.2.3. HBase

For HBase only one table was created that contains three Columnar Families and that stores only one version of each value inserted.

HBase splits data into regions¹² which are automatically and evenly distributed across the available region servers meaning that the programmer has nothing special to do to have all the regions distributed evenly. The memory configuration was the following: 500 MB of memory was allocated to each set of HBase processes on each node and 1GB of memory was allocated to each set of Hadoop processes on each node.

It is interesting to mention the architecture used for the HBase cluster as it is more complicated than a system where all the nodes are equal. One node was selected as the master for both HBase and HDFS levels, meaning that one node were running the HBase's master, a Zookeeper server as well as the Hadoop's namenode. Finally, each node including the master also runs an HBase's region server and a Hadoop's datanode.

3.2.4. MongoDB

For mongoDB, I used only one database that contains three collections. The memory management was left to mongoDB as it does not need to be set by hand. This is because mongoDB use memory mapped files for all disk I/O, therefore the caching management is left to the operating system.

With mongoDB, the data is distributed across the nodes following the chosen shard key as well as the size of the chunks. The shard key is the field of the document chosen as the one that will be used to split the range of all data into chunks. The values in the chosen field must be unique and there must be a strict ordering between those values. Then all the chunks will be automatically distributed evenly across the available shards in the cluster.

The architecture used for the mongoDB cluster was the following: the nodes were grouped by three into replica sets, each of them acting as a shard and each node also runs a mongos process that can receive and route requests to the corresponding shards. Each of the three first nodes of the cluster also runs a configuration server. It is important to note that in this configuration only a third of the cluster's nodes will serve both the read and write requests.

The two thirds left act only as backup if strong consistency is desired.

3.3. DATA SET

A new key space was created from the console with two columnar families. One of the columnar families holds the data and the second records the time for each execution. 100,000,000 Read write request was used in order to simulate what goes on in large scaled websites that handles a lot of read and write request. This is similar to large scaled websites like Google, twitter,, Amazon, yahoo, Facebook etc.

To create this simulation, I created a keyspace named mydata and two columnar families named data and time respectively.

The data consisted of 10 columns and 10 million Rows.

This was chosen so that 100 million read write request can be generated from the data set. This is a heavy load which structured very language (SQL) based database cannot handle because it is based on a single machine. Trying to use SQL to handle such would lead to several time cuts without getting result. This is made possible with a number NoSQL horizontally scaled machines.

The second columnar family consists of 2 columns and the number of rows created depends on the time the system is stabilized. The following php scripts were created, createdata.php, were codes for populating mydata column family was created, readwrite100million.php and readwrite200million.php scripts were also created to generate the 100 million reads and writes as well as 200 million reads and writes.

Tabular Representation

Data

Key-row	1st_item	2nd _item		9th_item
Key1	1st_item1	2nd _item1		9th_item1

Key2	1st_item2			9th_item2
Key3	1st_item3			9th_item3
:	:	:		
:	:	:		
Key10000000	1st_item10000000	2nd_item10000000		9th_item10000000

Table 3.1 Tabular Representation

Timer

Key-Row	Seconds
0	0
1	?
2	?
.	
.	
Nth Term	?

Table 3.2: Timer Table

Key-row	1st_item	2nd_item		9th_item
Key0	1 st _item0	2 nd _item0		9 th _item0

Table 3.3: Writer Table

WRITER

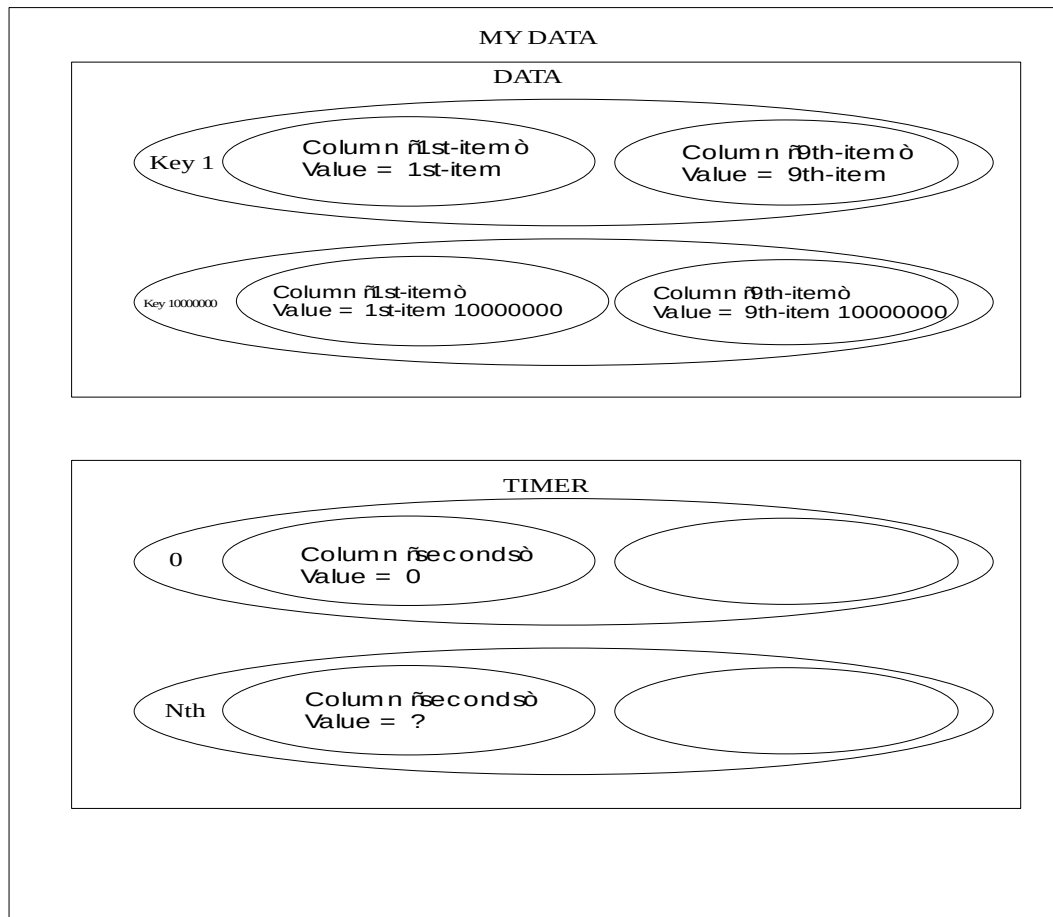


Fig. 3.1: The Data and Timer Diagram

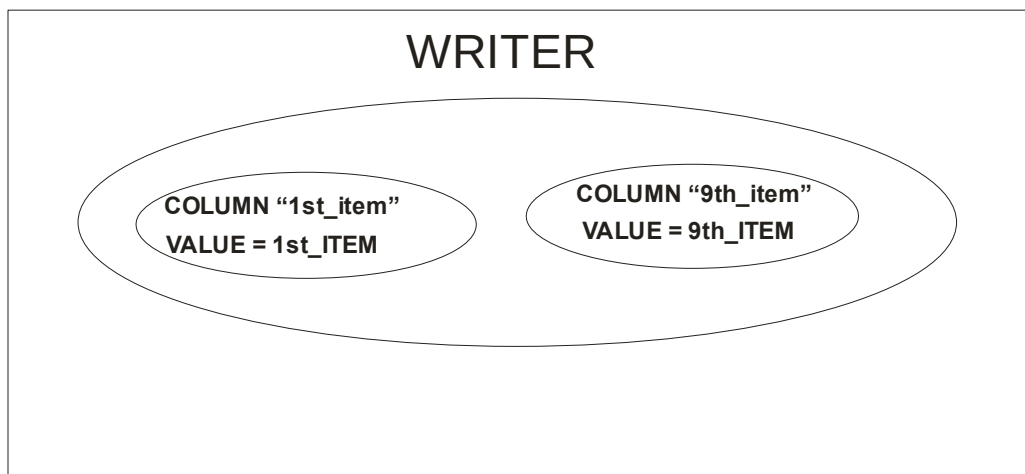


Fig. 3.2: The writer diagram

3.3.1. CREATING THE MYDATA KEYSPACE AND THE TWO COLUMN FAMILIES

When installing the NoSQL database, one node was used as the seed node this keeps configuration of the rest of the nodes on the cluster.

Taking Cassandra for example, open the bin directory from the console.

```
./cassandra-cli-h localhost
>create keyspace mydata;
>use mydata;
>create column family data;
>create column family writer;
>create column family time;
>exit;
```

The following script created a database keyspace named mydata, enabled the database and created two column families data and timer inside it. The timer column family had one key-row named '0' with one column name seconds created and a value of 0 assigned to it.

Unlike RDBMS (Relational database Management Systems), NoSQL is not schema structured. Different columns can be inserted during operation without being predefined.

3.3.2. Createdata.php. Script

This program script was developed to repopulate the data column family and give it 10 columns and 10 million rows.

Code

//call autoload script from the library in order to load its class.

```
require ('lib/phpcassa/autoload.php');
```

//select some classes to use

Use phpcassa\column family;

Use phpcassa\connection\connection pool;

//open the mydata keyspace;

```
$pool = new connection pool ("mydata");
```

//Now to open the data column family

```
$column_family = new column family ("spool; 'data');
```

//Now create 10 columns and 10 million rows

```
for ($k = 1; $k = < 10000000; $k++) {
```

```
    $key = "key." $k;
```

```
    $1st_item = "1st_item". $k;
```

```
    $2nd_item = "2nd_item". $k;
```

```
    $3rd_item = "3rd_item". $k;
```

```
    $4th_item = "4th_item". $k;
```

```
    $5th_item = "5th_item". $k;
```

```
    $6th_item = "6th_item". $k;
```

```
    $7th_item = "7th_item". $k;
```

```
    $8th_item = "8th_item". $k;
```

```
    $9th_item = "9th_item". $k;
```

```
$column-family—→insert ('$key', array ('1st -item' -> '$1st-item; '2nd - item =>
'$2nd - item; '3rd - item' -> '$3rd - item'; 4th - item' -> $4th - item; 5th -
item'-> $5th - item; 6th - item'-> $6th - item; 7th - item'-> $7th - item; 8th -
item'-> $8th - item; 9th - item'-> $9th - item')));}
```

```
//at the end of this for loop all 1000000 rows has been inserted into the 10
//columns.
```

```
// Now we will create the column family for writer.
```

```
$column_family = new column family ("spool; 'writer');
```

```
$column_family -> insert (key0 array (1st_item -> "1st_item0" ;
```

```
2nd_item -> "2nd_item0 "; 3rd_item -> "3rd_item0 ;
```

```
4th_item -> "4th_item0 "; 5th_item -> "5th_item0 ;
```

```
6th_item -> "6th_item0 "; 7th_item -> "7th_item0 ";
```

```
8th_item -> "8th_item0 "; 9th_item -> "9th_item0 " ; ))
```

```
?>
```

3.3.3. Readwrite100million.php Script

The aim of this script, is to read all the data that have been inserted into the data column family and also write them to the write column family. At the end of reading 10 columns and 10 million rows, it must have read 100 and written 100 million different data. This is to simulate the type of heavy operation faced by these large scaled websites thereby testing the time it takes each of these NoSQL model, to read them while being horizontally scaled.

When new nodes are attached, first they announce their arrival to the pre-existing nodes between the new and old nodes while these operations are taking place, these nodes still serve request sent to them by clients. But because these nodes are also busy doing load balancing between its clusters, the rate at which they respond to request continues to slow down for some NoSQL models this

takes longer time than others. After a certain time, the cluster gains stability and speeds up again. The 3 NoSQL involved here are the Cassandra, HBASE and MongoDB.

Fluctuations were measured as these models processes their request after which performance and stability are calculated. The script `readwrite100million.php` also writes to the column family timer at the end of each read. It takes record of how long it took for each read operation to complete once the script `readwrite100million.php` starts it continues until it is manually stopped. The timer column family was monitored from time to time until conclusion was taken that the cluster has gained stability. When the recent time intervals have no much difference from the previous time intervals, the script was stopped.

3.3.4. THE CODE (`readwrite100million.php`)

```
<? Php

//call thye necessary classes from library required ('lib/phpcasa/alltoload.php);

Use php cassa\connection\connection pool;

//get time for starting execution in seconds

$begintime = time ();

//connect to keyspace

$pool = new connection pool ("mydata");

// To read data from the data column family

For ($k = 1; $k <= 10000000; K + +){

//open the column family data

$column_family -> new ColumnFamily ($pool; 'data')
```

```

$key = "key". $k;

$1st_item = "1st- item". $key;

$2nd – item = "2nd – item" . $k;

$3rd – item = "3rd – item" . $k;

$4th – item = "4th – item" . $k;

$5th – item = "5th – item" . $k;

$6th – item = "6th – item" . $k;

$7th – item = "7th – item" . $k;

$8th – item = "8th – item" . $k;

$9th – item = "9th – item" . $k;

$column_family -> get ('$key');

//open the column family writer

$column family -> new ColumnFamily ($pool; 'writer')

$column_family -> insert ( $key, array( '1st_item'-> $1st_item ;

'2nd_item'-> $2nd_item ; '3rd_item'-> $3rd_item ;

'4th_item'-> $4th_item ; '4th_item'-> $4th_item ;

'5th_item'-> $5th_item ; '6th_item'-> $6th_item ;

'7th_item'-> $7th_item ; '8th_item'-> $8th_item ;

'9th_item'-> $9th_item ; ));

}

$endtime = time ();

```

```
$timetaken = $endtime - $begintime;
```

```
//now open the time insert time into the seconds column also insert the number  
of respective execution at its key-row.
```

```
$column-family = new ColumnFamily ('$pool' 'timer');
```

```
//now get the last inserted row key inserted and
```

```
//increment it by one for the next rowkey
```

```
$count = get —→ ('row-key');
```

```
$count = $count + 1;
```

```
$columnfamily = insert ('$count', array ('seconds' => '$timetaken'));
```

```
//return back to page
```

```
Header ('location: read100million.php');
```

```
exit;
```

```
?>
```

3.3.5. Readwrite200million.php Script

This script is the same as read100million.php but this time another loop, a nested loop of 2 counts is placed above the exiting one to make the read request execute twice thereby causing 200 million reads instead of 100 million reads

```
;
```

```
;
```

```
For ($m = 1 ; $m <= 2; $m ++ ) {
```

```
For ($k = 1; $k ≤ 10000000; $k ++){
```

```
$key = "key" . ('$key');
```

}

}

;

;

;

This script is used as the nodes are doubled again from (6 -12 nodes) to (12 – 24 nodes) and observations noted.

3.4. The Result Format

The result can be viewed from the time column family

It looks like the table below:

Key-Row	0	1	2	3	4	-	-	-
Seconds	0	50	60	58	55	-	-	-

Table 3.4: Result from time column family

The result for each is used to derive diagrams using matlab program.

These results are used to measure performance and scalability of each No.SQL model.

3.5. Program Flow

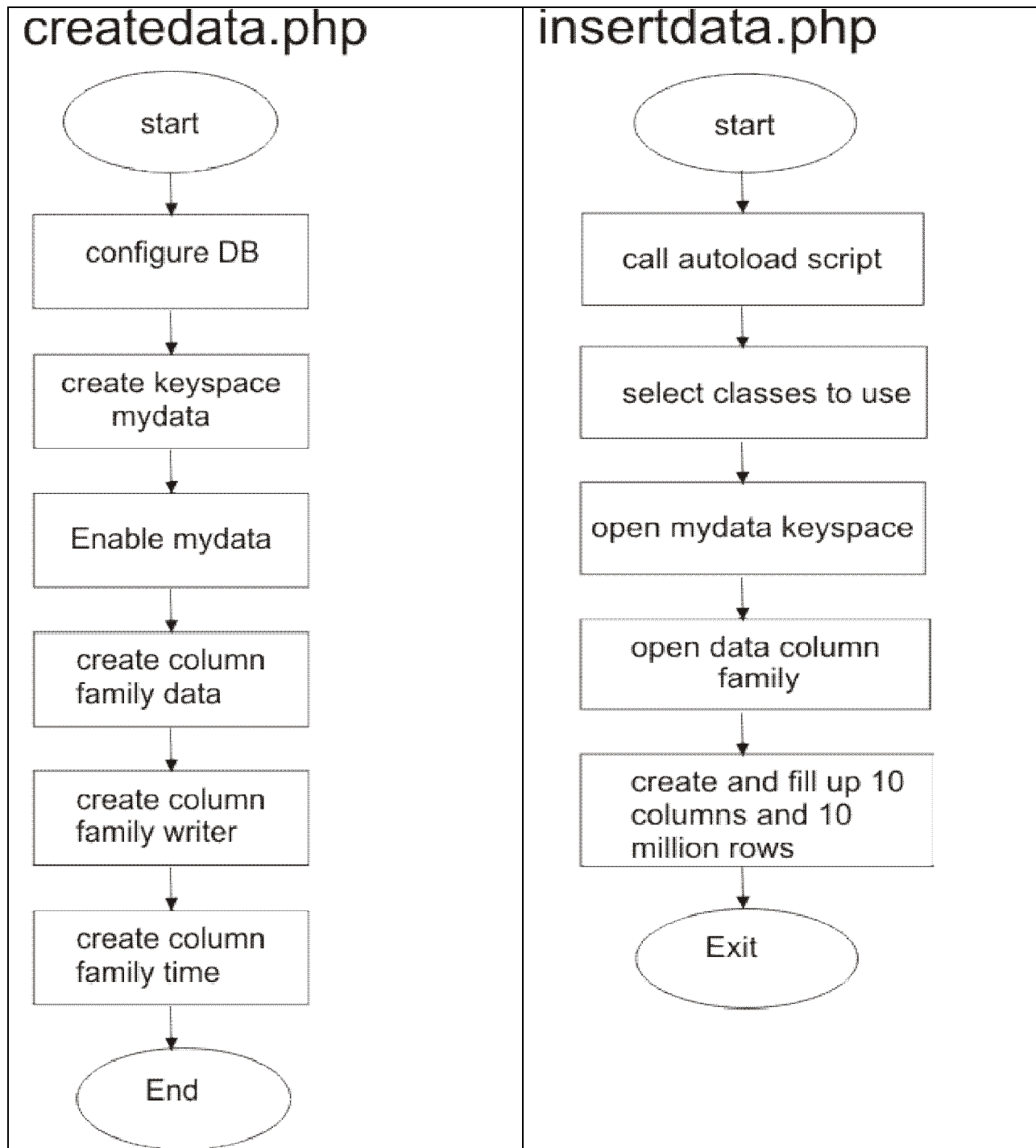


Fig. 3.3: Program flow diagram (createdata.php And insertdata.php)

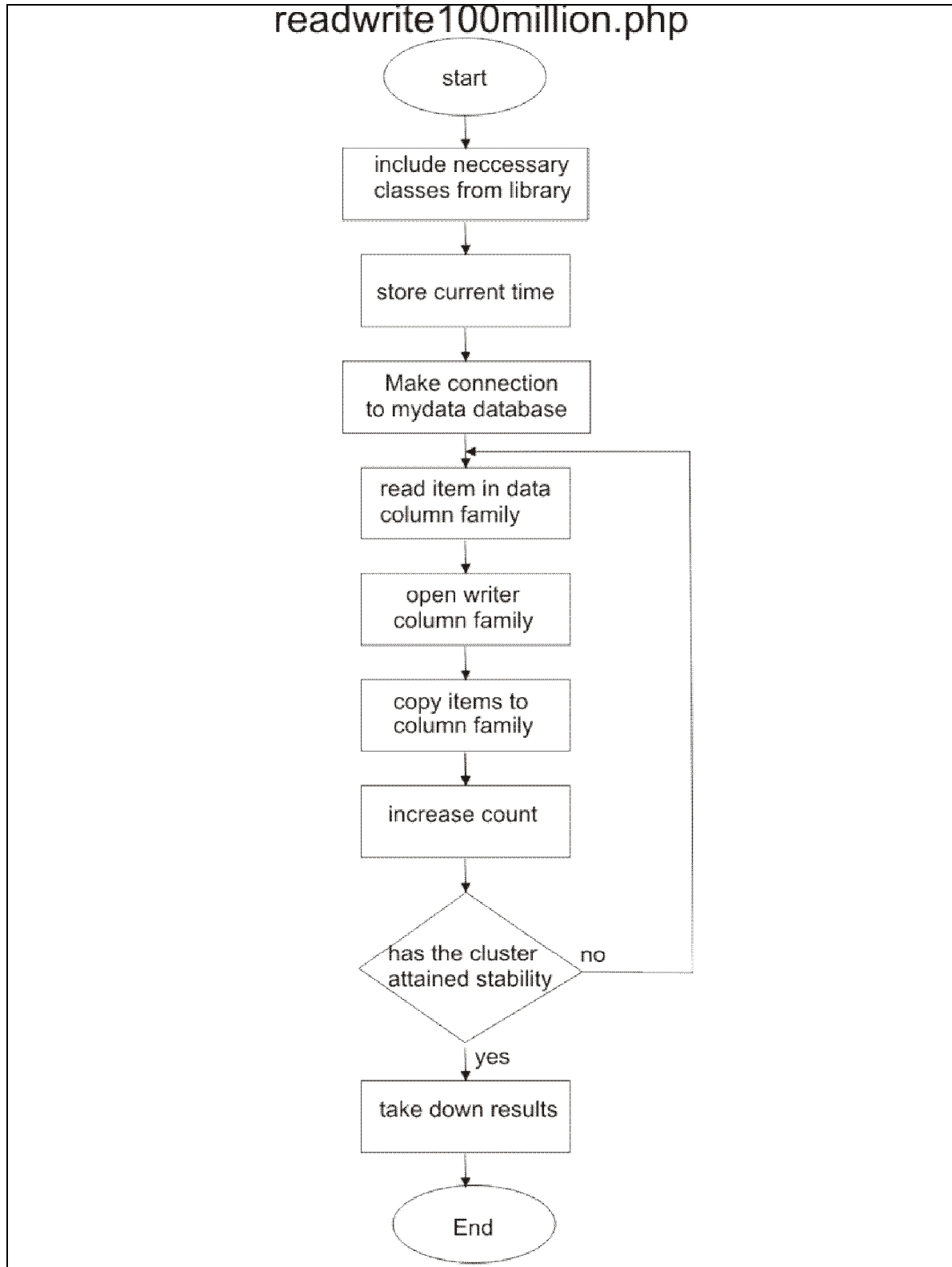


Fig 3.4: Program flow diagram (readwrite100million.php)

3.6. Network Configuration

This is based on testing these models from 6nodes to 12 nodes to 24 nodes. There by taking note of how they respond as clusters as it increases.

NETWORK DEVICE	IP ADDRESS
SWITCH 1	192.168.0.1
SWITCH 2	192.168.0.2
SWITCH 3	192.168.0.3
PC A	192.168.0.4
PC B	192.168.0.5
PC C	192.168.0.6
PC D	192.168.0.7
PC E	192.168.0.8
PC F	192.168.0.9
PC G	192.168.0.10
PC H	192.168.0.11
PC I	192.168.0.12
PC J	192.168.0.13
PC K	192.168.0.14
PC L	192.168.0.15
PC M	192.168.0.16
PC N	192.168.0.17
PC O	192.168.0.18
PC P	192.168.0.19
PC Q	192.168.0.20
PC R	192.168.0.21
PC S	192.168.0.22
PC T	192.168.0.23
PC U	192.168.0.24
PC V	192.168.0.25
PC W	192.168.0.26
PC X	192.168.0.27

Table 3.5: Network Diagram

3.7. Network diagrams

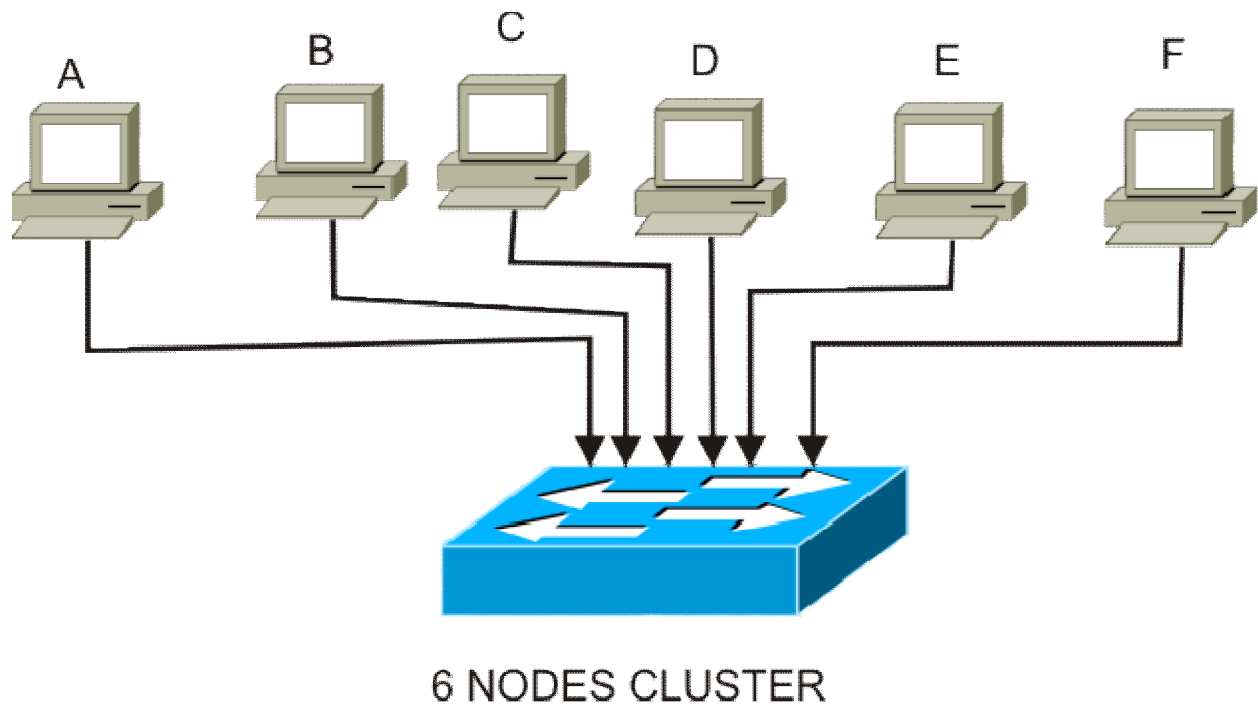


Fig 3.5: 6 nodes cluster

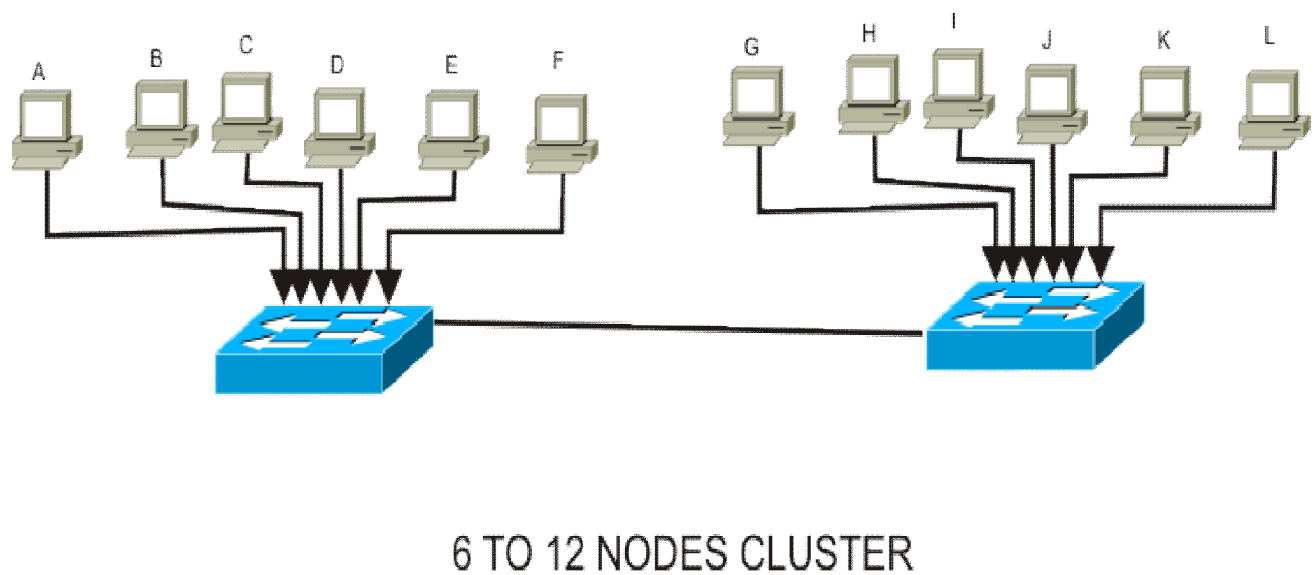
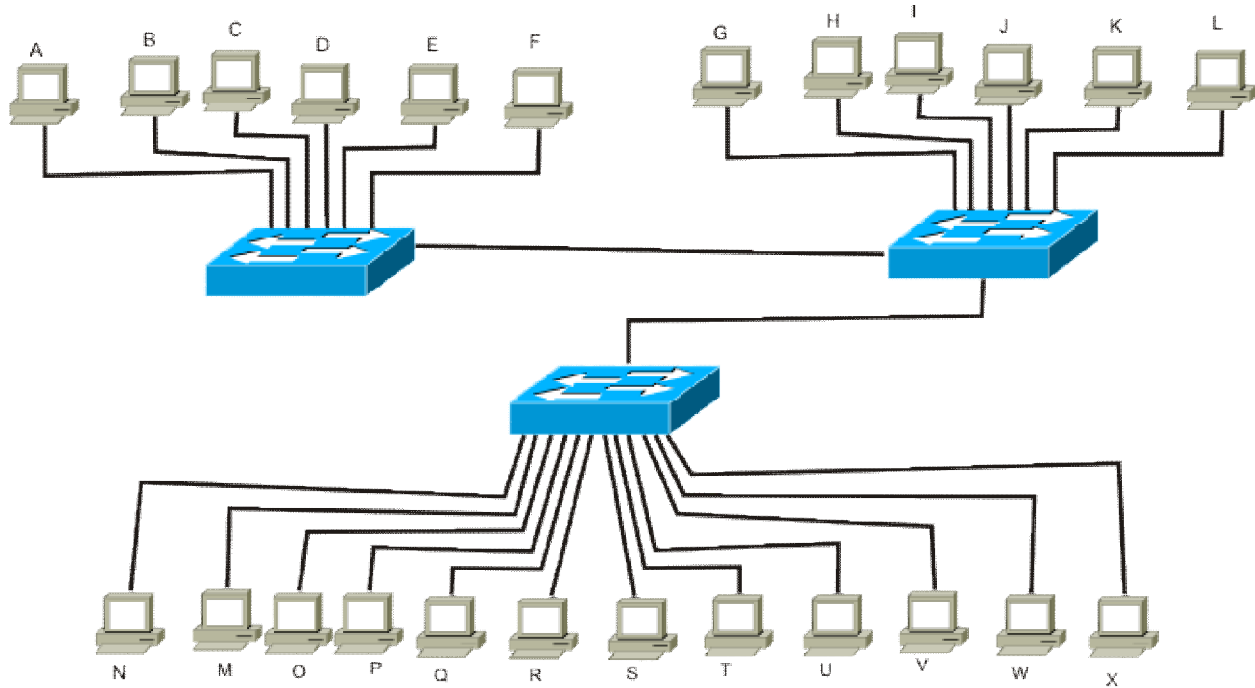


Fig 3.6: 6 -12 nodes cluster



12 TO 24 NODES CLUSTER

Fig 3.7: 12 - 24 nodes cluster

3.8. SOLVING LOAD BALANCING PROBLEM IN NoSQL

The total number of nodes I used in my cluster was 12 nodes. Drawing the ring we have

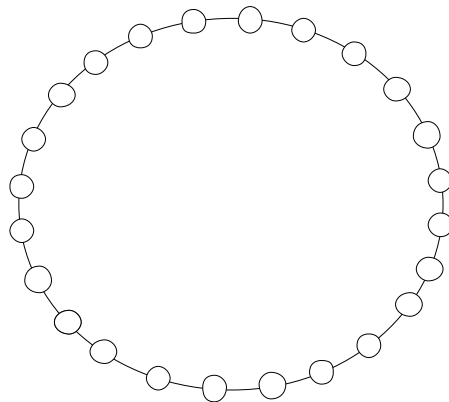
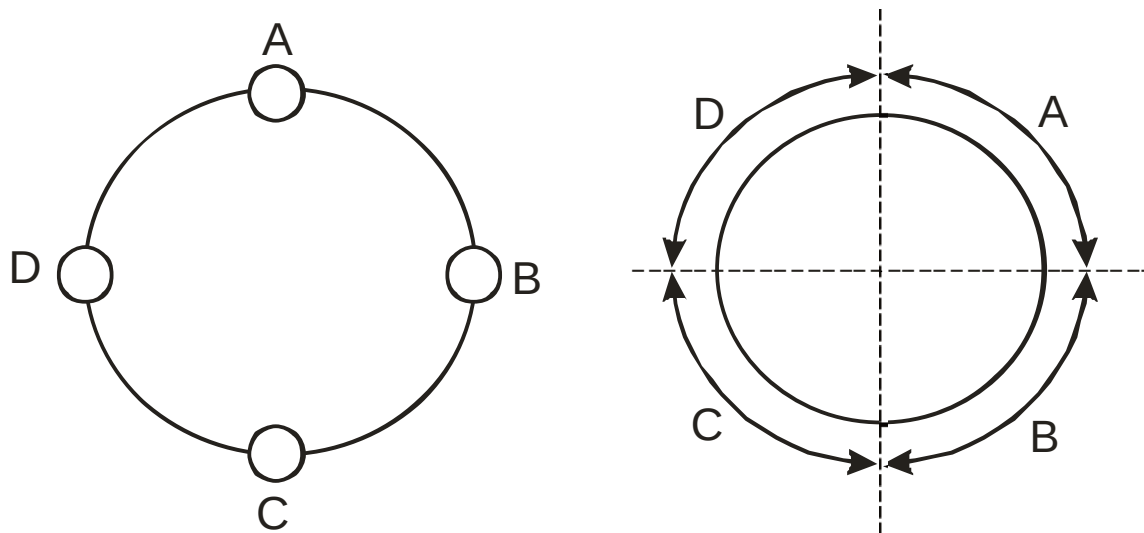


Fig 3.8: 24 nodes on a ring cluster

3.9. Experiment showing the effect of load balancing

Now let's use 4 hashed node on a cluster for an example

Fig 3.9: 4 hashed node on a cluster



Many NoSQL database models including Cassandra, scalaris etc distribute their load by first counting the number of nodes on the cluster then dividing the number of request by the number of nodes.

There is absolutely no problem with this when all nodes can process request at the same speed. But for economical purposes some people buy different types of servers according to the money available. Now the processing speed of these nodes vary but the NoSQL database still assigns the same amount of load for each of them, this becomes a problem.

This very problem I was able to solve by converting all my physical nodes to several virtual servers.

Let's first solve the problem above where we have four nodes A, B, C, and D.

If these nodes have the same processing speeds, there is no problem but problem when they have different request processing speed because they will still be assigned the same load distribution.

An Example

Assuming the following nodes can at this rate

A Processes 10 request per second

B Processes 20 request per second

C Processes 20 request per second

D Processes 30 request per second

Not minding their capabilities, the same amount of request is sent to them.

Assuming 8,000 request are sent,

$$\text{A will be assign } \frac{8000}{4} = 2000 \text{ request}$$

$$\text{B will be assign } \frac{8000}{4} = 2000 \text{ request}$$

$$\text{C will be assign } \frac{8000}{4} = 2000 \text{ request}$$

$$\text{D will be assign } \frac{800}{4} = 200 \text{ request}$$

Since there are 4 nodes, each is assigned

$$\text{Total number of request } \frac{800}{4} = 200 \text{ request each}$$

This means that it will take

A, 200 seconds to finish (since it processes 10/sec)

B, 100 Seconds to finish (since it processes 20/sec)

C, 100 Seconds to finish (since it processes 20/sec)

D, 66.66 Seconds to finish (since it processes 30/sec)

Since these 4 nodes are chained up as one replica just as it is said that a chain is as strong as its weakest point, it applies here. Before this replica set can write that the request has been executed, the slowest being node A must equally finish.

This means that it will take the cluster 200 seconds to complete the request causing delays at node B, C and D. Node B and C will experience a delay of 133 seconds this makes the system not to be effective Now to solve this problem we convert these nodes to virtual services. This is a process known as converting physical nodes to virtual servers.

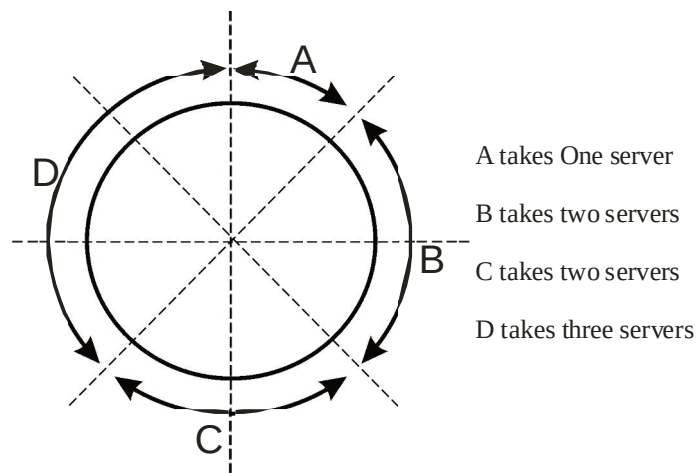
SOLUTION

Use the software physical to virtual servers converter to break each physical to a number of virtual servers Leave node A as it is, apply virtual conventions on servers B, C and D

Finally are here. A = 1 server, B = 2 servers (virtually converted) C = 2 servers (virtually converted) D = 3 servers (virtually converted)

Although we have a total of a physical server, we now have a total of 8 virtual servers and NoSQL load distributor now sees 8 servers instead of 4 and it will now distribute the request to 8 virtual servers.

Fig 3.10: The result



A Receives 1000 request

B Receives 2000 request

C Receives 2000 request

D Receives 3000 request

Remember a processes 10 request/sec

B Processes 20 request/sec

C Processes 20 request/sec

D Processes 30 request/sec

A will take A, $\frac{1000}{10} = 100$ seconds

A will take B, $2000/20 = 100$ seconds

A will take C, $2000/20 = 100$ seconds

A will take D, $3000/20 = 150$ seconds

All 4 nodes will finish at the same time. Now it takes the cluster only 100 seconds instead of 200 seconds making the system more efficient.

3.10. Application and result on my cluster

To experiment this practically on my cluster, I used 12 computers. All had the same RAM size of 1 Gig but first six had a CPU speed of 2.5GHz while the second six had a CPU of 4.0GHz. I divided them and mixed them up as 3 with 4.0GHz combined and later joined them to the remaining six node carrying out the same read request as before.

Now the first time I did this with converting to virtual servers then took down my result. I later repeated the experiment installing the Physical to virtual server software and breaking those with 4GHz to 4 virtual servers each and those with

2.5GHz I made to be 3 virtual servers then I repeated the same experiment and took down the results then I used matlab program to generate the diagrams for comparison and discovered that I have improved the efficiency of the cluster.

3.11. Microsoft System Center Virtual Machine Manager (VMM)

Microsoft System Center Virtual Machine Manager (VMM) allows you to convert existing physical computers into virtual machines through a process known as physical-to-virtual (P2V) conversion. VMM simplifies P2V by providing a task-based wizard to automate much of the conversion process.

You can download this software from [\[76\]](#).

3.12. RUNNING THE VIRTUAL MACHINE CONVERTER SOFTWARE

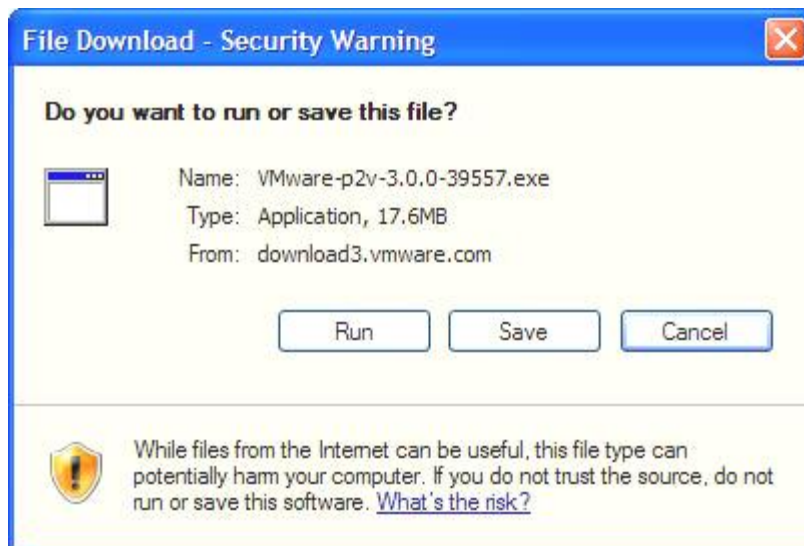


Fig 3.11: INSTALLATION SCREEN



Fig 3.12: Run Screen



Fig 3.13: VMware Converter

Follow the necessary steps to configure VMM to your test. VMM converts a single server into the number of virtual machines you desire.

CHAPTER FOUR

4.0. RESULTS ANALYSIS AND DISCUSSION

4.1. Results and Calculations

Below is the output from the program represented in diagrams.

--	--	--

CASSANDRA 6-12 NODES		HBASE 6-12 NODES		MongoDB 6-12 NODES	
1	60	1	48	1	75
2	70	2	38	2	52
3	60	3	40	3	50
4	65	4	44	4	48
5	100	5	44	5	46
6	90	6	45	6	52
7	150	7	46	7	52
8	125	8	42	8	50
9	110	9	44	9	54
10	130	10	48	10	48
11	75	11	80	11	54
12	90	12	45	12	56
13	100	13	48	13	100
14	80	14	45	14	155
15	100	15	40	15	100
16	50	16	42	16	50
17	50	17	44	17	54
18	50	18	42	18	56
19	50	19	42	19	54
20	50	20	42	20	50
				21	48
				22	50
				23	68
				24	68
				25	52
				26	46
				27	45
				28	44
				29	45

--	--	--

Fig 4.1: FROM 6 TO 12 NODES RESULTS

CASSANDRA 12 - 24 NODES		HBASE 12 - 24 NODES		MONGODB 12 - 24 NODES	
1	70			1	125
2	68			2	115
3	65	1	44	3	162
4	65	2	45	4	170
5	90	3	48	5	120
6	95	4	45	6	118
7	100	5	44	7	115
8	95	6	50	8	125
9	75	7	48	9	125
10	100	8	60	10	130
11	100	9	50	11	130
12	180	10	52	12	125
13	125	11	53	13	120
14	75	12	52	14	140
15	60	13	50	15	135
16	125	14	50	16	138
17	115	15	48	17	140
18	110			18	135
19	100			19	128
20	105			20	130
21	100			21	128
22	115			22	80
23	90			23	84
24	95			24	86
25	90			25	200
26	60			26	135
27	62			27	165
28	60			28	100
				29	105
				30	100
				31	130
				32	80
				33	95
				34	92
				35	90
				36	91
				37	90
				38	89

Fig 4.2 FROM 12 TO 24 NODES RESULTS

4.2. MATH LAB DIAGRAMS

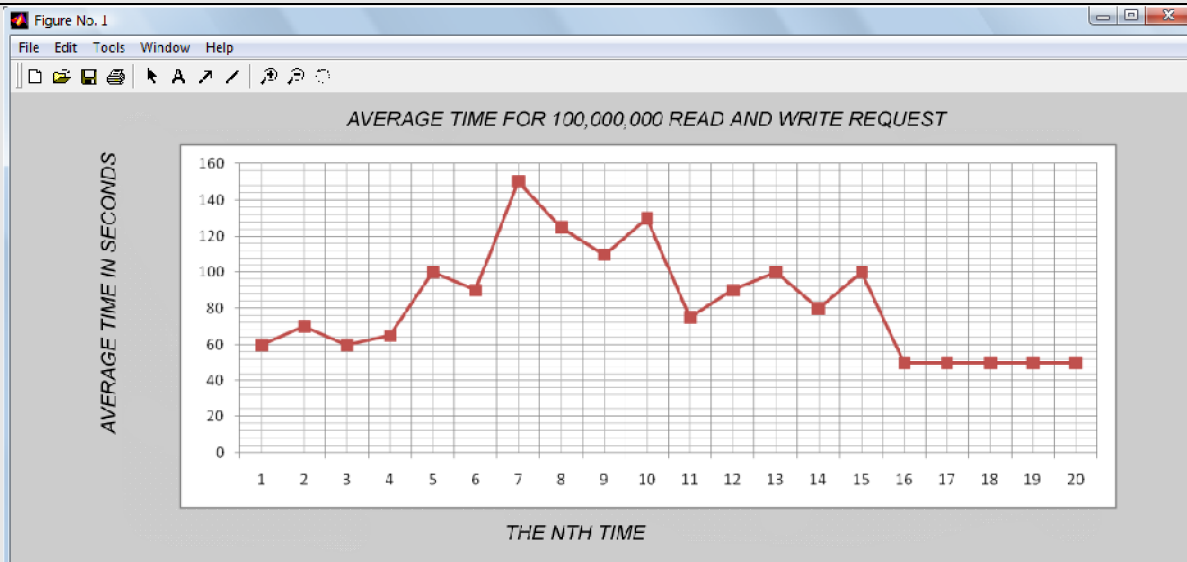
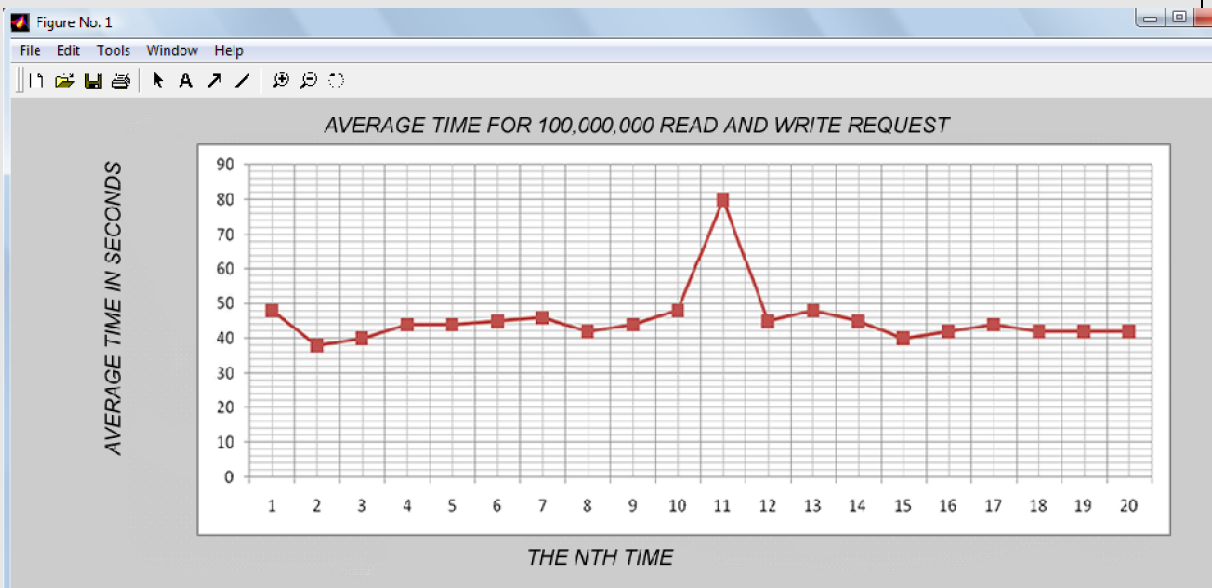
Fig 4.3: Cassandra (6 – 12 nodes)**Fig 4.4: HBASE (6 – 12 NODES)**

Fig 4.5: MongoDB (6 – 12 nodes)

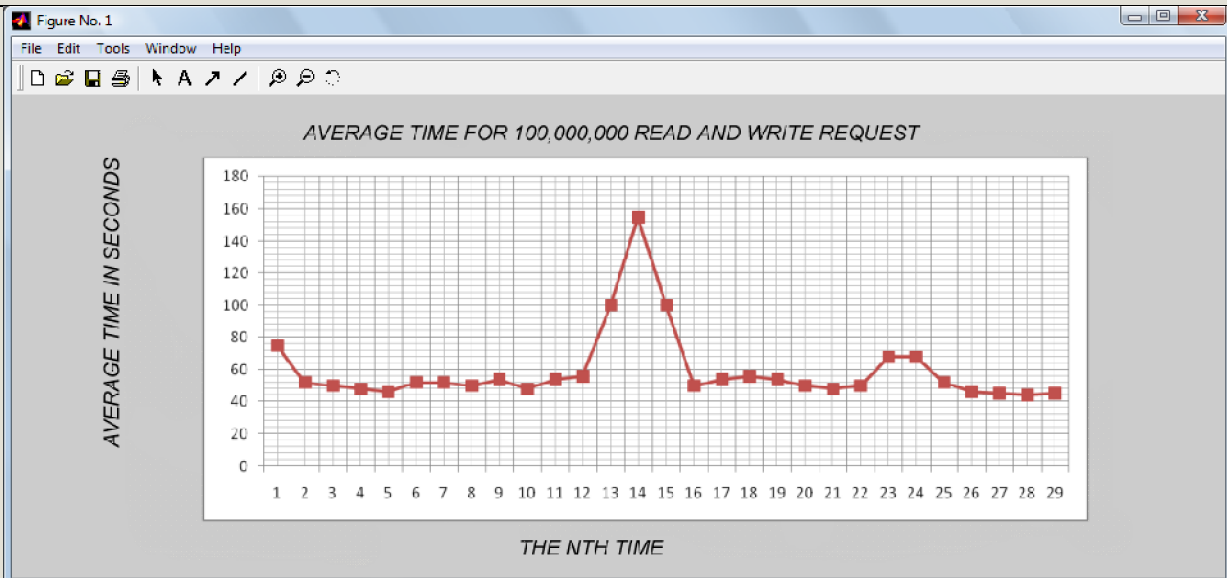


Fig 4.6: MongoDB (12 – 24 nodes)

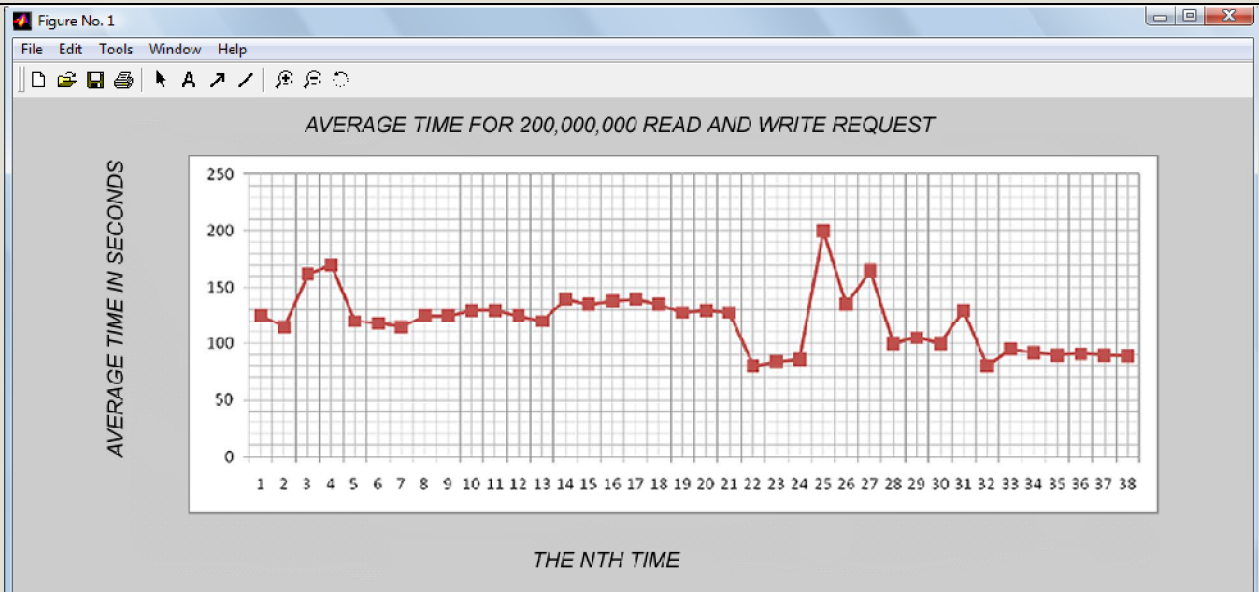
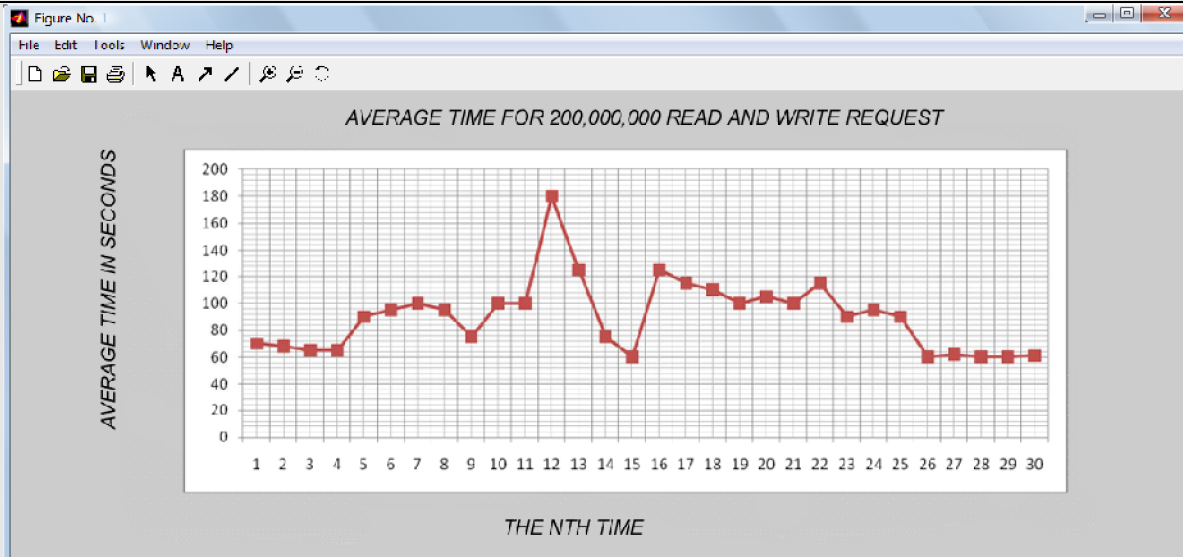
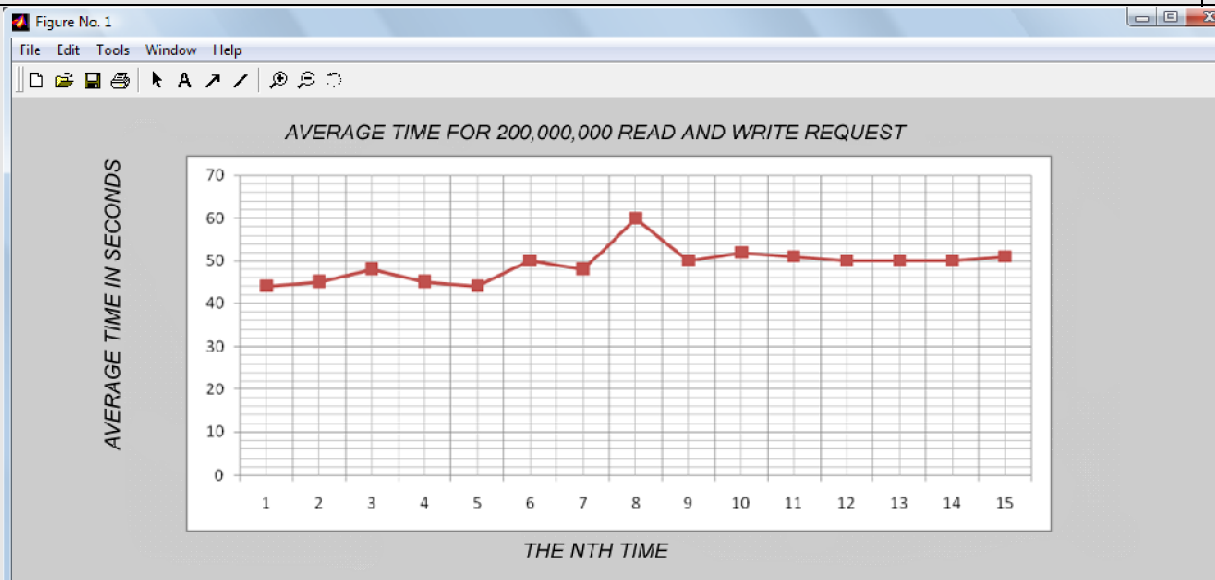


Fig 4.7: Cassandra (12 – 24 nodes)**Fig 4.8: HBASE (12 – 24 NODES)**

4.3. Stabilization Times Of The Database Models

6 – 12 nodes

Cassandra – (6 – 12 nodes) – 1,505 seconds = 25 mins

Hbase – (6 – 12 nodes) – 606 seconds = 10. 1 mins

Mongo DB – (6 – 12 nodes) – 1,408 seconds = 25 mins

12 – 24 nodes

Cassandra – (12 – 24 nodes) – 2408 secs = 40 mins

Hbase – (12 – 24 nodes) – 384 secs = 6.4 mins

MongoDB – (12 – 24 nodes) – 4,119 secs = 68.6 mins

Table 4.1: Stabilization time (in minute lower is better)

Database	Cluster old end news size	Date transfer time
Cassandra	6 to 12 nodes	25 mins
Hbase	6 to 12 nodes	10.1 mins

MongoDB	6 to 12 nodes	25 mins
Cassandra	12 to 24 nodes	40
Hbase	12 to 24 nodes	6.4
MongoDB	12 to 24 nodes	68.34

4.4. SCALABILITY

This is calculated as

$$\text{Scalability} = \frac{\text{Average } 2N - \text{Average } N}{\text{Average } N}$$

Average N

We have N to be the six nodes to 12 nodes when 100,000,000 read and write request was sent and 2N to be 12 – 24 nodes when 200, 000, 000 read request was sent

For Cassandra we have

Av. N = 100.3secs, Average 2N = 96.3 sec

$$\begin{aligned} \text{Scalability} &= \frac{100 - 96.3}{100} \\ &= -0.037 \end{aligned}$$

HBASE SCALABILITY

We have that

$$\text{Average N} = 46.6 \text{ secs}$$

$$\text{Average } 2N = 48 \text{ sec}$$

$$\text{Scalability} = 48 - 46.6 / 46.6$$

$$= 0.0300$$

MONGODB

$$\text{Average } N = 59.9 \text{ sec} = 60 \text{ sec}$$

$$\text{Average } 2N = 163.9 \text{ sec} = 164 \text{ sec}$$

$$\text{Scalability} = \frac{164 - 60}{60} = 1.73333$$

Scalability (lower is better)

Database	Closer older and newer size	Scalability
Cassandra	6 to 12 nodes	0.037
Hbase	6 to 12 nodes	0.0300
MongoDB	6 to 12 nodes	1.666

Table 4.2: Hbase has the best scalability property

4.5. Performance measurement

Getting performance means checking the average time for each database model for each number of read request. The average time shows the performance

CASSANDRA (6 – 12 NODES)

Total time before stabilization = 1,505 seconds

Number of time before reached = 15 times

Average time = 100.3 seconds

HBASE (6 – 12 NODES)

Total time before stabilization = 606 seconds

Number of time before reached = 13 times

Average time = 46.61 seconds

MONGODB (6 – 12 NODES)

Total before stabilization = 1,498 seconds

Number of time before reached = 25 times

Average time = 59.92 seconds

12 – 12 NODES MEASUREMENTS

CASSANDRA

Total before stabilization = 2408 seconds

Number of time before reached = 25 times

Average time = 96.32 seconds

HBASE (12 – 12 NODES)

Total before stabilization = 384 seconds

Number of time before reached = 8 times

Average time = 11.63 seconds

MONGODB (12 – 12 NODES)

Total before stabilization = 4,119 seconds

Number of time before reached = 33 times

Average time = 124.8 seconds

Performance in seconds (lower is better)

Database	Cluster size	No. of operation	Average time (sec)
Cassandra	6 to 12 nodes	100,000,000	100.3
Hbase	6 to 12 nodes	100,000,000	46.61
MongoDB	6 to 12 nodes	100,000,000	59.92
Cassandra	12 to 24 nodes	200,000,000	96.32
Hbase	12 to 24 nodes	200,000,000	11.63
MongoDB	12 to 24 nodes	200,000,000	124.8

Table 4.3: Performance in seconds

4.6. Comparison with results from Ecole Polytechnique De Louvain Computer Engineering Dept.

Below is the graphical representation of the result from measurements of the Ecole Polytechnique institution of Cassandra, Hbase and MongoDB compared with my own results.

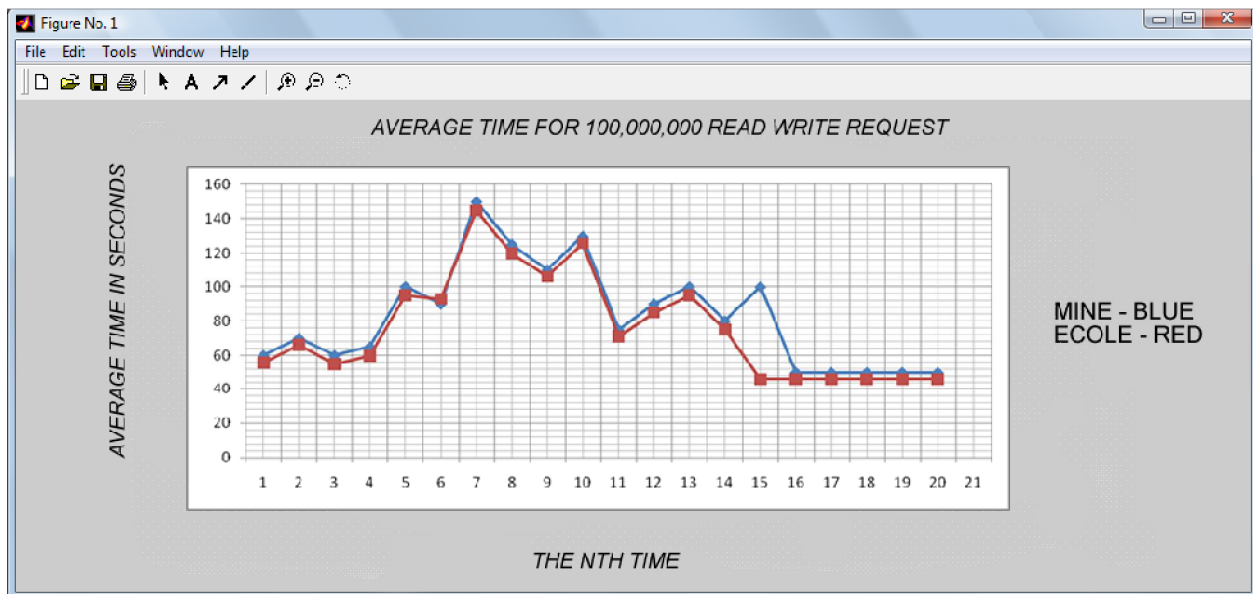


Fig 4.9: Compared Cassandra (6 - 12 nodes)

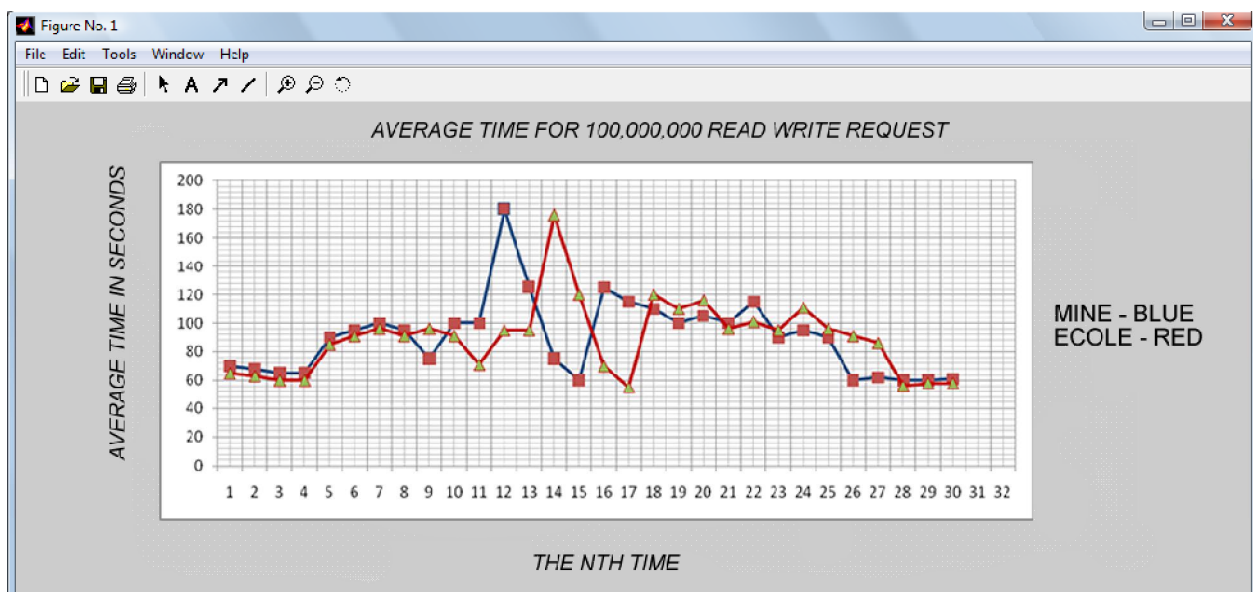


Fig 4.10: Compared Cassandra (12 - 24 nodes)

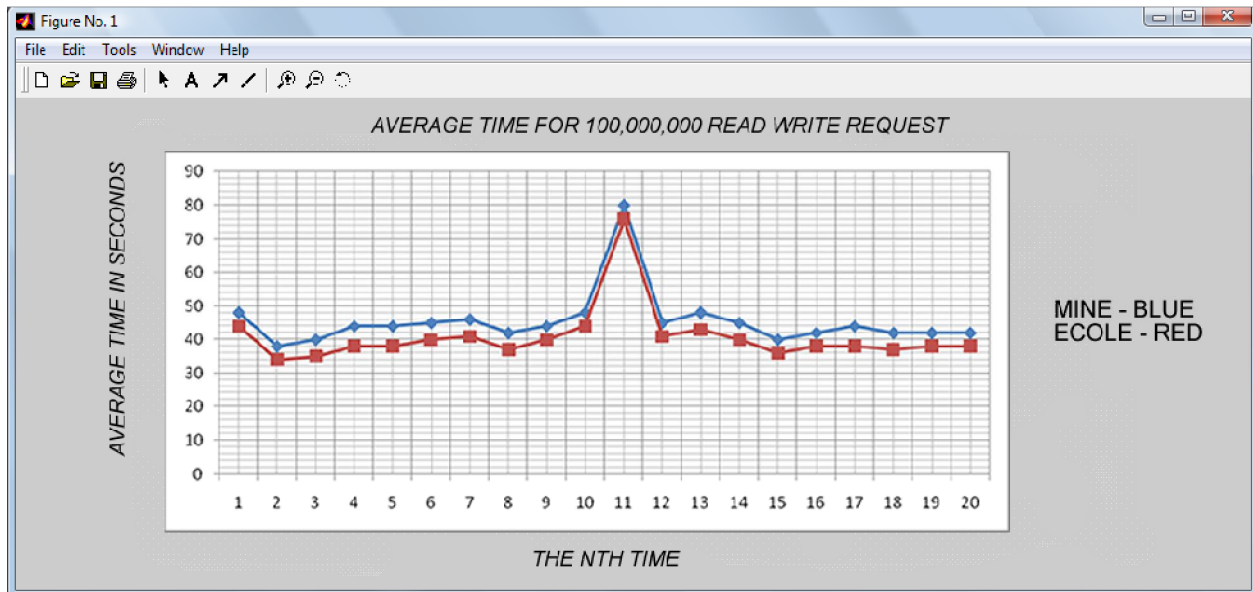


Fig 4.11: Compared Hbase (6 - 12 nodes)

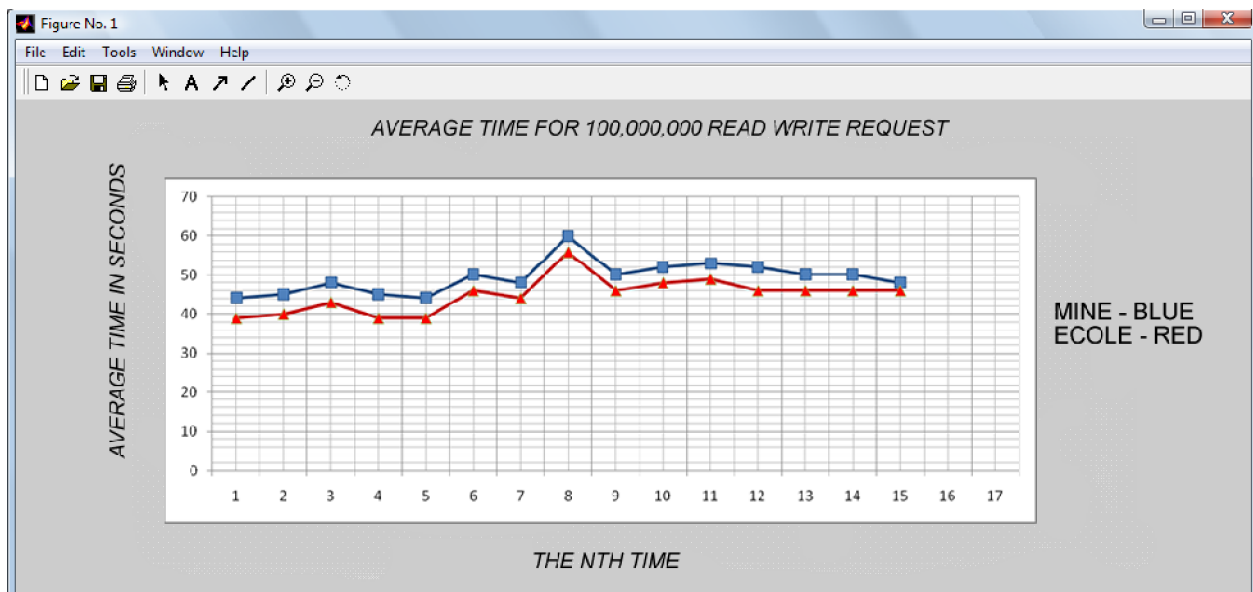


Fig 4.12: Compared Hbase (12 - 24 nodes)

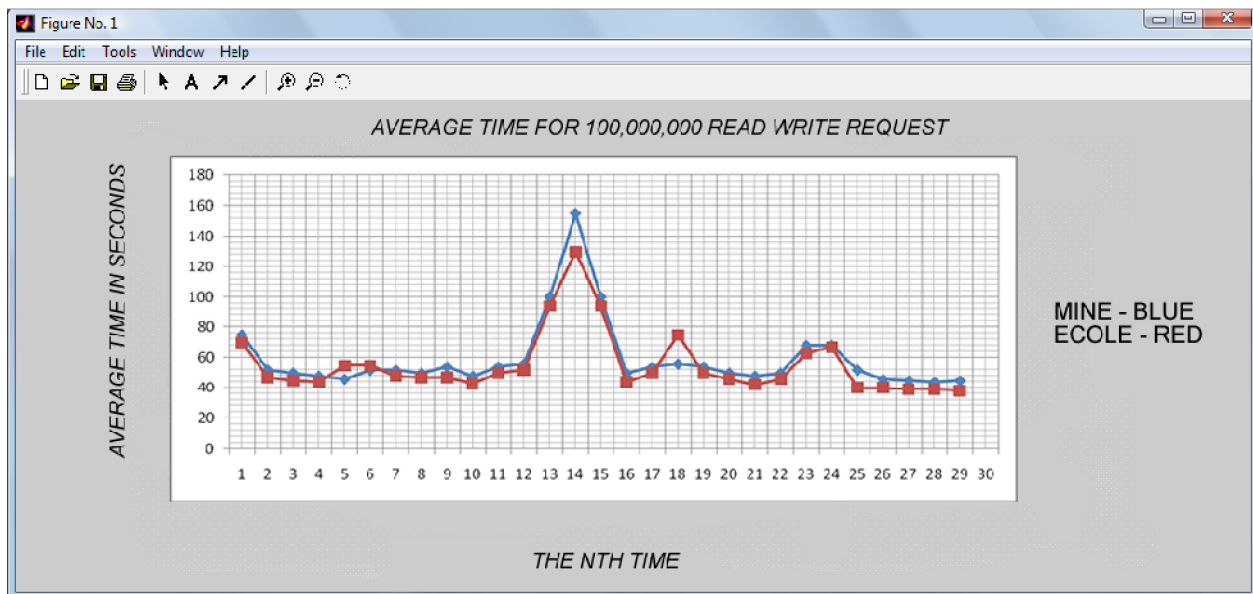


Fig 4.13: Compared Mongo (6 - 12 nodes)

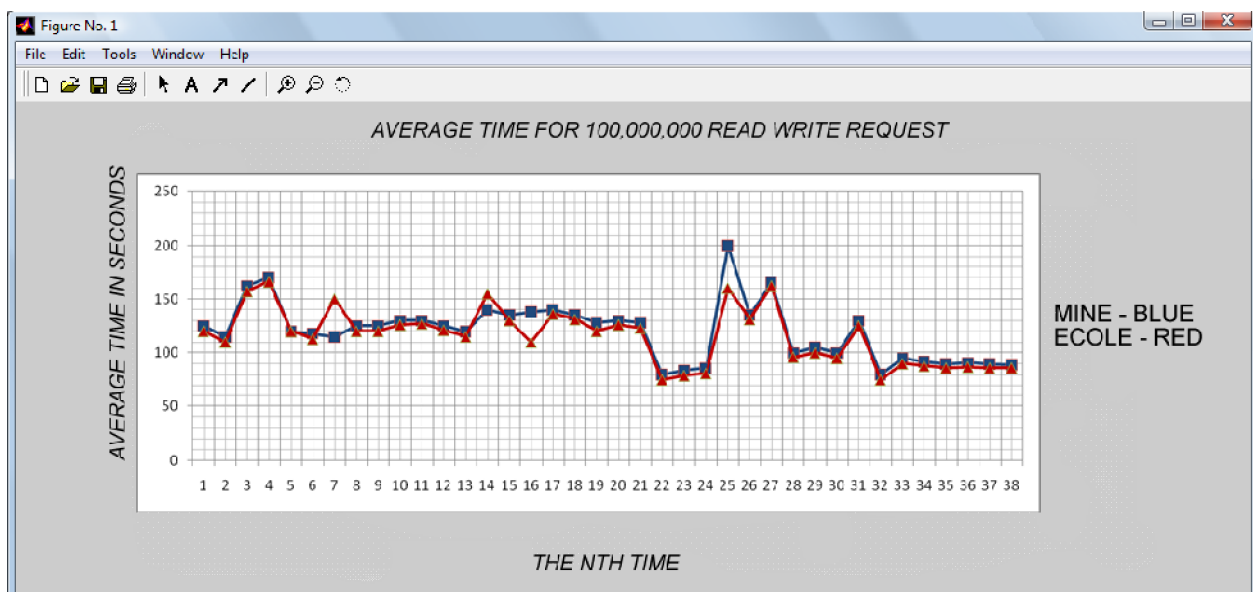


Fig 4.14: Compared MongoDB (12 - 24 nodes)

4.7. RESULTS OF PHYSICAL AND VIRTUAL NODES

THE RESULTS

PHYSICAL NODES		VIRTUAL NODES	
1	90	1	72
2	105	2	70
3	90	3	80
4	88	4	78
5	100	5	90
6	97	6	87
7	150	7	130
8	135	8	146
9	125	9	132
10	187	10	187
11	165	11	120
12	195	12	80
13	150	13	120
14	100	14	80
15	150	15	120
16	100	16	64
17	150	17	58
18	80	18	60
19	75	19	58
20	74	20	60
21	73	21	60
22	75		
23	75		

Fig 4.15: RESULTS OF PHYSICAL AND VIRTUAL NODES

Fig 4.16: Physical (6 – 12 nodes)

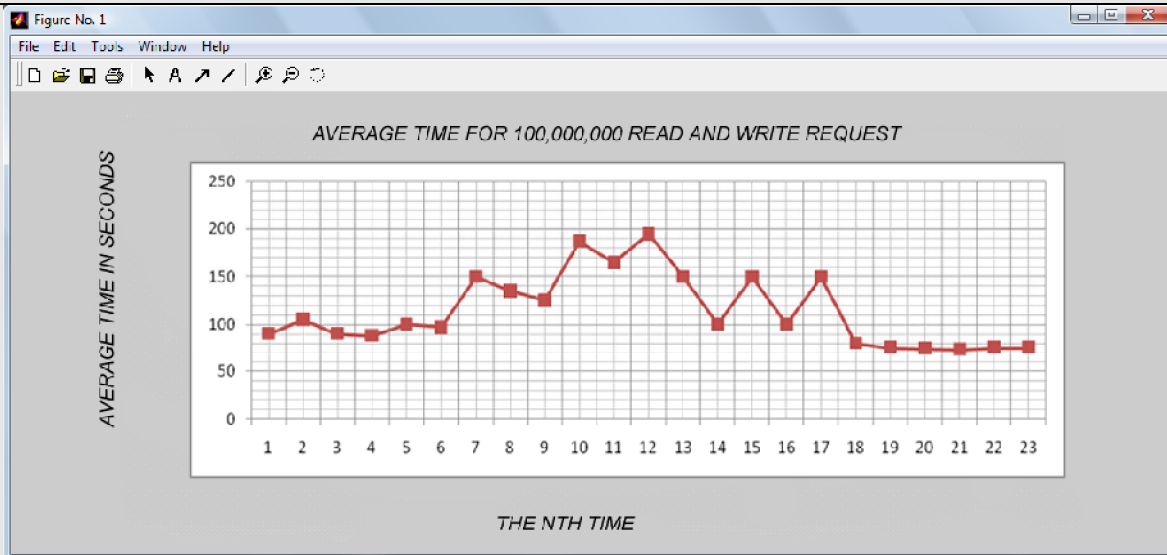


Fig 4.17: Virtual (6 – 12 nodes)

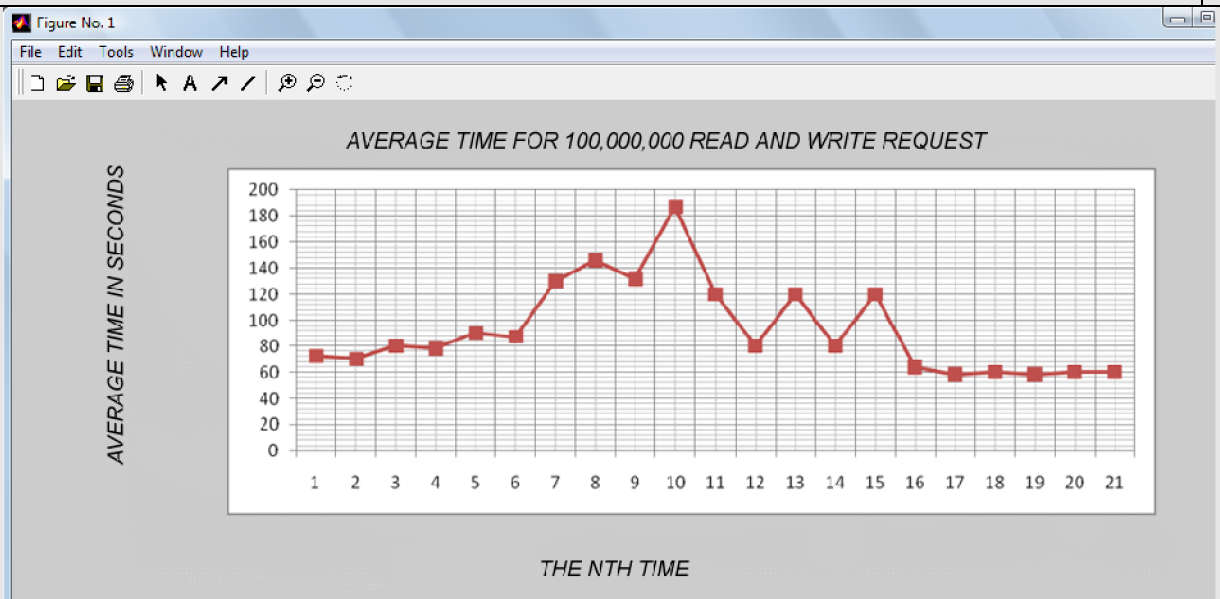
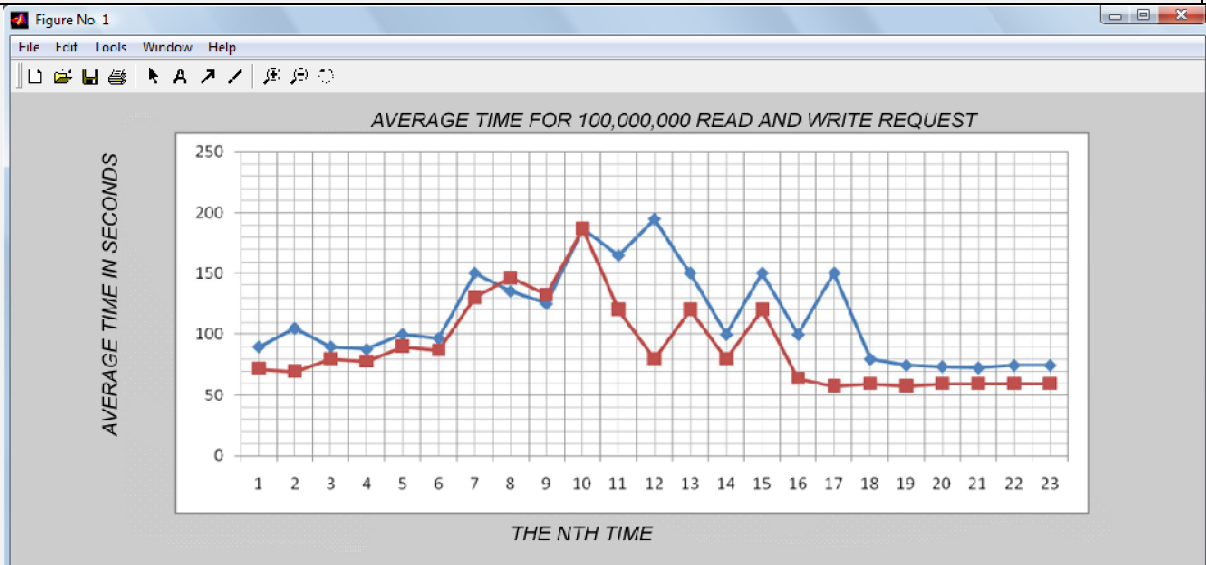


Fig 4.18: Comparison between Virtual & Physical (6 – 12 nodes)

Physical – BLUE. Virtual -- RED

From the tables and Matlab diagrams, we can see that the improvement is clear. Physical nodes (i.e. before the conversion) took longer time to stabilize with a total of 37 minutes 6 seconds. After the conversion, it only took 27 minutes 6 seconds to gain stability.

In the process before conversion, stabilization was seen after the 18th period but after the conversion, it gained stability after the 16th period.

The peak point of the physical nodes was 195 seconds while that of the virtual nodes was 187 seconds.

4.8. Performance and Stability Results

STABILITY

Here I calculated stability for the 12 Physical nodes (i.e. without any virtual conversion) then I also calculated for stability after the nodes has been converted.

PHYSICAL NODES STABILITY (Time taken for Physical nodes to stabilize)

Values in seconds before stabilization = $90 + 105 + 90 + 88 + 100 + 97 + 150 + 135 + 125 + 197 + 150 + 100 + 150 + 100 + 150 + 180 = 2,257$ seconds = 37.6 minutes.

VIRTUAL NODES STABILITY (Time taken for virtual nodes to stabilize)

Values in seconds before stabilization = $72 + 70 + 80 + 78 + 90 + 87 + 130 + 146 + 132 + 187 + 120 + 80 + 120 + 80 + 120 + 64 = 1,656$ = 27.6 minutes

PERFORMANCE RESULTS

PHYSICAL NODES

Total time before stabilization = 2,257 seconds = 37.6 minutes

Number of times before reached = 18 times

Average time = $2,257 / 18 = 125.38$ seconds.

VIRTUAL NODES

Total time before stabilization = 1,656 seconds = 27.6 minutes

Number of times before reached = 16 times

Average time = $1,656 / 16 = 103.5$ seconds.

AVERAGE PERFORMANCE AFTER STABILIZATION

Physical nodes => $(75 + 74 + 73 + 75 + 75) / 5 = 74.4$ seconds.

This is the time it takes it to complete the 100,000,000 read and write request

Virtual nodes => $(58 + 60 + 58 + 60 + 60) / 5 = 59.2$ seconds.

This is the time it takes it to complete the 100,000,000 read and write request.

4.9. RESULTS ANALYSIS

Analyzing the measurement results can be more difficult by the variability of the cluster performance under load before new nodes are bootstrapped / added.

The variability is very clear for Cassandra, HBase and mongoDB. The big variability in performance have different origins but I have made the assumption that all of them have the same immediate cause: the writing of at least one big file on the disk. This assumption is based on my observation of the logs and the fact that big writes are consuming a lot of I/O while they are always needed by the database. Those big writes are triggered by various events depending on the database:

- HBase: Big writes are triggered when compactions or disk flushes occur. The flushes occur when the memtable is full. The algorithm that triggers compactions is a little bit more complex [77]. A load constantly updating data will, sooner or later, trigger compactions and disk flushes.
- Cassandra: Big writes are also triggered when compactions or disk flushes occur. By default, a compaction process is started each time 4 SS Tables of the same size

are present on disk. An SS Table is written on disk each time a memtable, which stores all the data written into Cassandra, is full [78] and therefore triggers a flush to disk. A load constantly updating data will, sooner or later, trigger compactions and disk flushes.

- MongoDB: In the case of mongoDB, big writes are only triggered when disk pre-allocations occur. MongoDB uses the mmap function provided by the operating system instead of implementing the caching layer itself, meaning that it is the OS itself that decides when to flush. By default, the flushes occur every 60 seconds, but this can be configured. MongoDB pre-allocates big files (2GB when the database is already filled with several gigabytes) when it needs new storage space instead of increasing the size of existing files. In practice, disk pre-allocation occurs when a lot of data is written to a given node like during the insertion of the data or when a new node bootstraps and starts receiving a lot of chunks. This implies that mongoDB should not write big files often during standard operations, but will as soon as new nodes are bootstrapped or big inserts are done.

It is important to note that compaction is part of normal database operation that is needed both when handling client requests and when handling bootstrapped nodes so I made no effort to remove the compaction cost from our measurements. It is important to note that the only requests that will be slowed down by the writing of big files will be the ones sent to nodes currently writing those big files. If the number of nodes increases, the probability to send requests to a node doing a lot of I/O decreases. Indeed, looking at graphs for Cassandra and HBase, I observed that the overall performance is more stable for bigger clusters.

The technical choice taken by mongoDB to make small but frequent disk flushes leads to less variability in performance than Cassandra. One could wonder what would be the cause of the variability noticed at the beginning of the chart for mongoDB as no new nodes were bootstrapped as of that time. The cause of this big variation in performance is also due to big files being written to disk, triggered by the fact that during the insertion, some nodes stored more chunks than the other and only started to distribute them across the cluster during the start of the test. Except for this exception, mongoDB's performance is more stable on this infrastructure than Cassandra.

The variability noticed in HBase performances is quite different from Cassandra even if their technical choices are close. By default the memtable's size of Cassandra is 64MB and HBase is 256MB, leading to more frequent flushes and compactions for Cassandra but on the other hand, the compactions are also made on smaller files for Cassandra. The effect of compactions is only visible on smaller number of nodes. This could be because the number of nodes turns bigger and the effect of the compaction impacted a smaller number of requests.

Coming to the case of mongoDB, there are no results for mongoDB going from 24 upwards. This is due to several problems encountered with mongoDB during the insertion of the data. Starting with a cluster of size 12, mongoDB processes started to crash because of segmentation faults that caused data corruption, even with the journaling enabled. This problem was temporarily fixed by increasing the maximum number of files that can be opened by the mongoDB processes [79]. But for 24 nodes, the segmentation faults were back with another problem. Eight threads were used to insert the data, each of them making its requests to a

different mongo's router process, but all the writes were done on the same replica set. The elected master of this replica set was moving the chunks to other replica sets but not as fast as it was creating them, leading to a disk full on the master and at this point all the inserts stopped instead of starting to write chunks on other replica sets. Further research and discussion on the mailing list of mongoDB8 have led to the conclusion that all those problems came from the fact that integers were used as primary keys instead of hash [61]. It is therefore important to know that it can lead to this kind of problems when big sequential writes are done from multiple clients whose cumulated bandwidth is bigger than the replica set's master handling the writes.

Scalability

The scalability performance shows that, for cluster sizes 6 and 12, both Cassandra and HBase manage to keep nearly constant performance after the linear increase of all the parameters, with Cassandra even increasing performance a little. On the other hand, mongoDB suffers a 78% decrease in performance after the linear increase of all the parameters and therefore shows poor scalability.

Application of Virtual servers

The experiment carried out here (i.e. conversion of physical nodes into virtual nodes) shows that in system where the processing speeds of servers are not equal, the services of virtual nodes are required. My experiment shows that stability is achieved quicker and performance is even better with virtual nodes rather than physical nodes.

CHAPTER FIVE

5.0. CONCLUSION AND RECOMMENDATION

NoSQL database is the answer for large scaled websites today and it is very important to know about these database models. The theoretical study, the practical considerations and the measurements conducted for this study led me to a better understanding of the implications of the choices and approaches taken by the databases. Using this knowledge, I define a few use cases, that are specific kind of problems and the best associated technical choices. I also listed a few recommendations to get the most of the databases from the research and practical work I did.

5.1. Use cases

The use cases are defined following the kind and the amount of data to store as well as the kind of load and its outlooks. This information is summarized into concrete applications and associated audiences. Note that only the databases considered as stable have been taken into account.

5.2. Website with medium to high traffic

From my observations, the best current NoSQL tool to run a website that does not need to store real “Bigdata” would be mongoDB. Indeed, as it has been shown in the results, it can be problematic to use large clusters with mongoDB and the scalability of the storage performance is not the best one.

In practice, a lot of highly frequented websites do not need to store vast amounts of data and in this case, mongoDB has several advantages. First, its document

oriented data model can ease the design of the database by having the possibility to store every web page as a single document even if each page has its own specifications. Meaning that every content appearing on a web page, can be stored into the same document. That avoids using JOIN operations over multiple tables and that ensures optimal performances.

Indeed, only one random seek on the hard disk would be mandatory to fetch the whole page. Secondly, it can provide directly the performances of a caching layer, without needing to have software to do complex tuning, thanks to the fact that mongoDB is directly using the caching layer of the operating system.

Note that the two previous arguments are also valid for Riak and Voldemort if the application building the web pages structures itself the content stored inside those databases. But it is important to retain that this work can be non trivial and directly leads to the next big advantage of mongoDB for this kind of application: its powerful query language. With mongoDB, developers used to the SQL language will find that most of the computation can still be done into the database itself instead of having to compute everything by hand on the client side like with key/value stores.

Finally it is important to mention that mongoDB is one of the database that is the most easy to configure and to use. There is not much tuning that can be done to achieve the best performance and even if that could looks like a drawback to some system administrators. This is a huge time saver and that simplify the management of the database. If the database gets slow, use more powerful hardware or add more nodes and mongoDB will take advantage of it on its own.

The APIs bindings are available in a wide variety of languages and the concepts easy to learn for anyone with a little experience with databases.

5.3. File storage in the cloud

Previously people store their information only in files and continue heap files upon another causing lost of information, delays and so many other things. Nowadays more and more services like Dropbox, Box.net or Ubuntu One provide end users with the possibility to store a subset of their personal files on distant servers. The biggest advantages are the availability of the files from almost any device connected to the internet and the assurance that those files will never be lost.

Any database that could be used to solve this storage problem must meet the following criteria:

1. A really scalable storage able to store “Bigdata” growing almost continuously
2. The possibility to store big binary files
3. Good latency but not necessarily excellent
4. A convenient way of structuring users’ data as well as segmenting the whole data set into user’s data set
5. The certainty that the user’s data will never be lost

The results of the measurements have shown that mongoDB [80] do not provide a good scalability and therefore do not meet the first criteria.

It is not recommended [81] to use HBase as storage for big binary blobs but this is exactly the kind of work HDFS has been designed for. It is also possible to use Cassandra to solve this problem, but it would require a little bit more work on the client side. Indeed it is not recommended [82] to store files bigger than 64Mb into a single column but there are no limitations to the number of column a row can count, therefore it would suffice to split the big files into 64 chunks and store each of them in a separate column inside the same row. Note that it is possible to use Cassandra for this kind of application because the level of concurrency will always be small. The advantage with the Cassandra solution over the HDFS one is the fact that the cluster is fully distributed with no single point of failure but in practice both would be usable. It is also interesting to note that there is new work in progress on HDFS to avoid having this single point of failure [83].

5.4. Conclusions

The measurements of the storage elasticity are enlightening of the true elasticity for the selected databases. The theoretical analysis of the expected elasticity and scalability has been proved correct regarding to the advantage of the systems that don't need to move all the data like HBase. For the systems that need to move the data, the impact on the time needed to finish all the transfers has also been observed. As expected this impact grows bigger with the size of the cluster. On the other hand, the measurements have also highlighted a few unforeseen consequences of the technical choices. The fact that new mongoDB nodes start faster to serve requests seemed like an advantage because it would spread the load faster on a bigger number of nodes. But in practice, the fact that those new nodes start serving requests as soon as they have downloaded a complete chunk, implies that they will also be serving requests while they pre-allocate big files on the disk to make room for the next chunks to come. The pre-allocations will consume a lot of I/O and therefore degrade the performances of the node.

The goal of those measurements is to observe in practice what is the real elasticity and scalability of those databases because it is very likely to observe unforeseen behaviors compared to the polished version given by the databases themselves. In practice the measurements have shown surprising behaviors for Cassandra, with an observed super linear scalability gain between 12 and 24 nodes.

Concerning mongoDB, the surprises come from the bad scalability and the stability problems. The bad scalability is surprising because the requests were correctly distributed among the shards that were serving a fair share of the data set. Again, more research would be necessary to discover what the cause of this loss of performance is. The overloading of a single shard at insert time was even more surprising because there is absolutely no indication in the documentation that choosing an integer as sharding key could result in this kind of behavior.

Finally, as the classification proposal has shown, the NoSQL databases selected for this study have made very different technical choices. Therefore it would be simplistic to say that one of the databases is the best in every situation. Those databases rather tend to be solutions to different and very specific problems. Choosing one of them for a given problem implies to understand well what would be the requirements in terms of consistency, queries, data model and distribution.

5.5. Further work

Study and comparism of modern NoSQL models for large scaled websites is a very vast and broad area as the world's data size continues to increase every day further research is needed. This Master's thesis has only been scratching the surface of what is possible to learn about the real performance, stability and scalability of distributed databases.

There are still numerous unexplained behaviors that would require a lot more time and budget to be fully understood. It is also important to note that the practical orientation of this work has only been focused on scaling up elasticity

and scalability. In real applications it is also very important to know what the consequences of an eventual scale down of the infrastructure are. Indeed, more and more organizations are using cloud instances instead of real servers because of the cost, the short availability of new instances and the fact that those instances are paid on a per hour basis. Therefore, it would be very interesting to be able to scale up and down the infrastructure to follow the real load applied by the users. But this optimal scenario is only feasible if the scaling up and scaling down properties of the selected software components are known.

It will be important to try out these two things and see the outcome that will be observed. Firstly, increasing the number of nodes to try to reach new and currently hidden bottlenecks and secondly extending those measurements to other databases coming from both the NoSQL world and the RDBMS world. The comparison with the old solutions that are the RDBMS like MySQL cluster could be very interesting as this solution is currently used by some big players like Twitter and facebook (who still use MySQL cluster for some parts of their infrastructure) with strong requirements both in term of performances and storage capacity.

Finally, the infrastructure used for the measurements has big advantages in terms of price and availability but it is a shared infrastructure, meaning that there are additional sources of variability.

REFERENCES

1. <http://insidehpc.com/2006/07/14/what-is-a-clustered-database/>
2. <http://engineering.twitter.com/2>
3. Fitzpatrick, Brad et al.: *Protocol*. March 2010.
4. JBoss Community Team: *JBoss Cache*. 2010. –<http://jboss.org/jbosscache>
5. Ho, Ricky: *NOSQL Patterns*. November 2009. Blog post of 2009-11-15.
6. <http://horicky.blogspot.com/2009/11/NoSQL-patterns.html>
7. Bemmann, Dennis: *Skalierbarkeit, Datenschutz und die Geschichte von StudiVZ*. January 2012.
8. Ho, Ricky: *NOSQL Patterns*. November 2009. Blog post of 2009-11-15.
9. Lipcon, Todd: *Design Patterns for Distributed Non-Relational Databases*. June 2009. –Presentation of 2009-06-11.
10. Wiki article, version 37 of 2010-10-21.
11. <http://www.mongodb.org/display/DOCS/Sharding>
12. MongoDB team: *MongoDB 1.6 Released*. August 2010. – Blog post of 2010-08-05.
<http://blog.mongodb.org/post/908172564/mongodb-1-6-released>
13. Ho, Ricky: *NOSQL Patterns*. November 2009. – Blog post of 2009-11-15.
14. *STOC '97: Proceedings of the twenty ninth annual ACM symposiums on Theory of computing*. New York, NY, USA: ACM, 1997, p. 654–663
15. <http://pdos.csail.mit.edu/chord/>
16. <http://labs.google.com/papers/gfs-sosp2003.pdf>
17. White, Tom: *Consistent Hashing*. November 2007. Blog post of 2007-11-27.
18. White, Tom: *Consistent Hashing*. November 2007. Blog post of 2007-11-27
19. Lipcon, Todd: *Design Patterns for Distributed Non-Relational Databases*. June 2009. Presentation of 2009-06-11.
20. Kreps, Jay et al.: *Project Voldemort – Design*. 2010.
21. <http://highscalability.com/blog/2009/10/28/and-the-winner-is-mysql-or-memcached-ortokyo-tyrant.html>

22. Kreps, Jay et al.: *Project Voldemort – Design*. 2012.
23. <http://project-voldemort.com/design.php>
24. <http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelationaldatabases>
25. O’Neil, Patrick ; Cheng, Edward ; Gawlick, Dieter ; O’Neil, Elizabeth:
26. <http://labs.google.com/papers/bigtable-osdi06.pdf>
27. <http://www.stanford.edu/ouster/cgi-bin/papers/ramcloud.pdf>
28. http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principlesbehind-the-NoSQL-alternatives.html
29. David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 2007
30. Facebook, Cassandra: A structured storage system on a P2P Network
31. www.facebook.com/note.php?note_id=24413138919
32. In Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP ’03, pages 29–43, New York, NY, USA, 2003. ACM
33. mongoDB, Frontpage. www.mongodb.org
34. A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2), 2008
35. In Proceedings of the 7th symposium on Operating systems design and implementation, OSDI ’06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association
36. Eric A. Brewer. Towards robust distributed systems (abstract). In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, PODC ’00, pages 7–, New York, NY, USA, 2000. ACM
37. Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33:51–59, June 2002
38. <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>
39. 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 Thomas C. Bressoud and M. Frans Kaashoek, editors, SOSP, pages 205–220. ACM, 2007
40. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler
41. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST ’10, pages 1–10, Washington, DC, USA, 2010
42. IEEE Computer Society
43. <http://wiki.apache.org/hadoop/ZooKeeper/HBaseUseCases>
44. The apache hbase book, chapter 11. data model
45. <http://hbase.apache.org/book.html#datamodel>
46. The apache hbase book, chapter 7. hbase and mapreduce.

- <http://hbase.apache.org/book.html#mapreduce>
47. Apache hbase, acid semantics. <http://hbase.apache.org/acid-semantics.html>
 48. Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system
 49. SIGOPS Oper. Syst. Rev., 44:35–40, April 2010
 50. Cassandra wiki, memtablesstable.
<http://wiki.apache.org/cassandra/MemtableSSTable>
 51. Cassandra wiki, datamodel. <http://wiki.apache.org/cassandra/DataModel>
 52. Cassandra wiki, api. <http://wiki.apache.org/cassandra/API>
 53. Cassandra wiki, architecture overview.
<http://wiki.apache.org/cassandra/ArchitectureOverview>
 54. <http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/>
 55. <http://www.mongodb.org/display/DOCS/Schema+Design>
 56. JSON FrontPage, Introducing JSON
 57. <http://www.json.org/>
 58. mongoDB documentation, Advanced Queries.
<http://www.mongodb.org/display/DOCS/Advanced+Queries>
 59. mongoDB documentation, MapReduce.
<http://www.mongodb.org/display/DOCS/MapReduce>
 60. <http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1>
 61. Kristina Chodorow. Scaling MongoDB. O'Reilly Media, 2011
 62. Basho wiki, what is riak? <http://wiki.basho.com/What-is-Riak%3F.html>
 63. Basho blog, schema design in riak - introduction.
<http://blog.basho.com/2010/03/19/schema-design-in-riak---introduction/>
 64. Zuse-Institut Berlin, Scalaris project details.
<http://www.zib.de/en/pvs/projects/details/article/scalaris.html>
 65. Scalaris, FrontPage. <http://code.google.com/p/scalaris/>
 66. Scalaris wiki, Users Developers Guide May 26 2010.
<http://code.google.com/p/scalaris/wiki/UsersDevelopersGuide>
 67. Scalaris wiki, How To Replica Placement. January 2012
<http://code.google.com/p/scalaris/wiki/HowToReplicaPlacement>
 68. Voldemort Project, Frontpage. <http://project-voldemort.com/>
 69. Voldemort Project, Design. <http://projectvoldemort.com/design.php>
 70. TPC, FrontPage. www.tpc.org

71. Benchmarking cloud serving systems with YCSB
72. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, SoCC, pages 143–154. ACM, 2010
73. <http://www.cisco.com/en/US/products/hw/switches/ps628/ps627/index.html>
74. <http://www.cisco.com/en/US/docs/switches/lan/catalyst2950/hardware/quick/guide/2950gsg2.html>
75. Matlab Primer R2012b (Nineteenth Edition September 2012)
76. <http://technet.microsoft.com/en-us/evalcenter/cc793138.aspx>
77. <http://www.outerthought.be/blog/465-ot.html>
78. <http://wiki.apache.org/cassandra/MemtableSSTable>
79. IRC channel #mongodb
80. <http://www.mongodb.org/display/DOCS/GridFS+Specification>
81. <http://blog.rapleaf.com/dev/2008/03/11/matching-impedance-when-to-use-hbase/>
82. [http://wiki.apache.org/cassandra/FAQ#large file and blob storage](http://wiki.apache.org/cassandra/FAQ#large_file_and_blob_storage)
83. <http://www.slideshare.net/huguk/hdfs-federation-hadoop-summit2011>
84. <http://en.wikipedia.org/wiki/SQL>
85. [http://en.wikipedia.org/wiki/Network switch](http://en.wikipedia.org/wiki/Network_switch)

APPENDIX

PROGRAM CODES

Database Script

```
./cassandra-cli-h localhost  
>create keyspace mydata;  
>use mydata;  
>create column family data;  
>create column family writer;  
>create column family time;  
>exit;
```

Createdata.php Script

```
//call autoload script from the library in order to load its class.  
require ('lib/phpcassa/autoload.php');  
  
//select some classes to use  
Use phpcassa\column family;  
Use phpcassa\connection\connection pool;  
  
//open the mydata keyspace;
```

```

$pool = new connection pool ("mydata");

//Now to open the data column family

$column_family = new column family ("spool; 'data'");

//Now create 10 columns and 10 million rows

for ($k = 1; $k = < 10000000; $k++) {

$key = "key." $k;

$1st-item = "1st_item". $k;

$2nd-item = "2nd _ item". $k;

$3rd-item = "3rd _ item". $k;

$4th-item = "4th _ item". $k;

$5th-item = "5th _ item". $k;

$6th-item = "6th _ item". $k;

$7th-item = "7th _ item". $k;

$8th-item = "8th _ item". $k;

$9th-item = "9th _ item". $k;

$column-family—→insert ('$key', array ('1st-item' -> '$1st-item; '2nd-item =>
'2nd-item; '3rd-item' -> '$3rd-item; 4th-item' -> $4th-item; 5th-item' -> $5th-item; 6th-item' -> $6th-item; 7th-item' -> $7th-item; 8th-item' -> $8th-item; 9th-item' -> $9th-item));}

//at the end of this for loop all 1000000 rows has been inserted into the 10
//columns.

// Now we will create the column family for writer.

$column_family = new column family ("spool; 'writer'");

```

```
$column_family -> insert (key0 array (1st_item -> "1st_item0" ;
2nd_item -> "2nd_item0 " ; 3rd_item -> "3rd_item0 ;
4th_item -> "4th_item0 " ; 5th_item -> "5th_item0 ;
6th_item -> "6th_item0 " ; 7th_item -> "7th_item0 " ;
8th_item -> "8th_item0 " ; 9th_item -> "9th_item0 " ; ))
?>
```

Readwrite100million.php Script

```
<? Php

//call the necessary classes from library required ('lib/phpcasa/alltoload.php);

Use php cassa\connection\connection pool;

//get time for starting execution in seconds

$beginntime = time ();

//connect to keypace

$pool = new connection pool ("mydata");

// To read data from the data column family

For ($k = 1; $k <= 10000000; K ++){

//open the column family data

$column_family -> new ColumnFamily ($pool; 'data')

$key = "key". $k;

$1st_item = "1st- item". $key;

$2nd - item = "2nd - item" . $k;
```

\$3rd – item = “3rd – item” . \$k;

\$4th – item = “4th – item” . \$k;

\$5th – item = “5th – item” . \$k;

\$6th – item = “6th – item” . \$k;

\$7th – item = “7th – item” . \$k;

\$8th – item = “8th – item” . \$k;

\$9th – item = “9th – item” . \$k;

\$column_family -> get (‘\$key’);

//open the column family writer

\$column family -> new ColumnFamily (\$pool; ‘writer’)

\$column_family -> insert (\$key, array(‘1st_item’-> \$1st_item ;

‘2nd_item’-> \$2nd_item ; ‘3rd_item’-> \$3rd_item ;

‘4th_item’-> \$4th_item ; ‘4th_item’-> \$4th_item ;

‘5th_item’-> \$5th_item ; ‘6th_item’-> \$6th_item ;

‘7th_item’-> \$7th_item ; ‘8th_item’-> \$8th_item ;

‘9th_item’-> \$9th_item ;));

}

\$endtime = time ();

\$timetaken = \$endtime - \$begintime;

//now open the time insert time into the seconds column also insert the number of respective execution at its key-row.

\$column-family = new ColumnFamily (‘\$pool’ ‘timer’);

```
//now get the last inserted row key inserted and
//increment it by one for the next rowkey
$count = get —→ ('row-key');
$count = $count + 1;
$columnfamily = insert ('$count', array ('seconds' => '$timetaken'));
//return back to page
Header ('location: read100million.php');
exit;
?>
```

Readwrite200million.php Script

```
<? Php
//call the necessary classes from library required ('lib/phpcasa/alltoload.php);
Use php cassa\connection\connection pool;
//get time for starting execution in seconds
$beginntime = time ();
//connect to keyspace
$pool = new connection pool ("mydata");
// To read data from the data column family
For ($k = 1; $k <= 20000000; $k++){
//open the column family data
$column_family -> new ColumnFamily ($pool; 'data')
```

```

$key = "key". $k;

$1st_item = "1st- item". $key;

$2nd – item = "2nd – item" . $k;

$3rd – item = "3rd – item" . $k;

$4th – item = "4th – item" . $k;

$5th – item = "5th – item" . $k;

$6th – item = "6th – item" . $k;

$7th – item = "7th – item" . $k;

$8th – item = "8th – item" . $k;

$9th – item = "9th – item" . $k;

$column_family -> get ('$key');

//open the column family writer

$column family -> new ColumnFamily ($pool; 'writer')

$column_family -> insert ( $key, array( '1st_item'-> $1st_item ;

'2nd_item'-> $2nd_item ; '3rd_item'-> $3rd_item ;

'4th_item'-> $4th_item ; '4th_item'-> $4th_item ;

'5th_item'-> $5th_item ; '6th_item'-> $6th_item ;

'7th_item'-> $7th_item ; '8th_item'-> $8th_item ;

'9th_item'-> $9th_item ; ));

}

$endtime = time ();

```

```
$timetaken = $endtime - $begintime;
```

```
//now open the time insert time into the seconds column also insert the number  
of respective execution at its key-row.
```

```
$column-family = new ColumnFamily ('$pool' 'timer');
```

```
//now get the last inserted row key inserted and
```

```
//increment it by one for the next rowkey
```

```
$count = get —→ ('row-key');
```

```
$count = $count + 1;
```

```
$columnfamily = insert ('$count', array ('seconds' => '$timetaken'));
```

```
//return back to page
```

```
Header ('location: read200million.php');
```

```
exit;
```

```
?>
```

Readwrite400million.php Script

```
<? Php
```

```
//call the necessary classes from library required ('lib/phpcasa/alltoload.php);
```

```
Use php cassa\connection\connection pool;
```

```
//get time for starting execution in seconds
```

```
$begintime = time ();
```

```
//connect to keyspace
```

```
$pool = new connection pool ("mydata");
```

```
// To read data from the data column family
```

```

For ($k = 1; $k <= 40000000; $k++){
//open the column family data
$column_family -> new ColumnFamily ($pool; 'data')

$key = "key". $k;
$1st_item = "1st- item". $key;
$2nd - item = "2nd - item" . $k;
$3rd - item = "3rd - item" . $k;
$4th - item = "4th - item" . $k;
$5th - item = "5th - item" . $k;
$6th - item = "6th - item" . $k;
$7th - item = "7th - item" . $k;
$8th - item = "8th - item" . $k;
$9th - item = "9th - item" . $k;
$column_family -> get ('$key');
//open the column family writer
$column_family -> new ColumnFamily ($pool; 'writer')
$column_family -> insert ( $key, array( '1st_item' -> $1st_item ;
'2nd_item' -> $2nd_item ; '3rd_item' -> $3rd_item ;
'4th_item' -> $4th_item ; '4th_item' -> $4th_item ;
'5th_item' -> $5th_item ; '6th_item' -> $6th_item ;
'7th_item' -> $7th_item ; '8th_item' -> $8th_item ;

```

```

'9th_item'-> $9th_item ; ));

}

$endtime = time ();

$timetaken = $endtime - $begintime;

//now open the time insert time into the seconds column also insert the number
of respective execution at its key-row.

$column-family = new ColumnFamily ('$pool' 'timer');

//now get the last inserted row key inserted and

//increment it by one for the next rowkey

$count = get —→ ('row-key');

$count = $count + 1;

$columnfamily = insert ('$count', array ('seconds' => '$timetaken'));

//return back to page

Header ('location: read400million.php');

exit;

?>

```