BRAC
UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGEENIRING

12-12-2012

"Investigating Cloud Data Storage"

Sumaiya Binte Mostafa (ID – 08301001)
Firoza Tabassum (ID – 09101028)

BRAC University

SUPERVISOR: Dr. Mumit Khan

INVESTIGATING CLOUD DATA STORAGE

By

SUMAIYA BINTE MOSTAFA

FIROZA TABASSUM

A report
submitted in the partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science & Engineering of
BRAC University

December 2012

BRAC UNIVERSITY

FINAL READING APPROVAL

Thesis Title: Investigating Cloud Data Storage

Date of Final Presentation: 12 December, 2012

The final reading approval of the thesis is granted by Dr. Mumit Khan, Supervisor.

Supervisor

_____

Dr Mumit Khan
Professor and Chairperson,
Department of Computer Science and Engineering,
BRAC University

ACKNOWLEDGEMENT

We would like to thank our supervisor Dr. Mumit Khan for his guidance and support he gave during this exercise. His inspiration and encouragement made it easier for us to finish the thesis work properly in proper time.

ABSTRACT

A cloud database is a database that typically runs on a cloud computing platform. Of the databases available on the cloud, traditional data model is SQL-based. The recent trend is to move on to NOSQL data model. Now, the question is which database approach is better to choose in this era of 'Big Data'? SQL databases are difficult to scale, meaning they are not natively suited to a cloud environment, although cloud database services based on SQL are attempting to address this challenge. On the other hand, NOSQL databases are built to service heavy read/write loads and are able scale up and down easily, and therefore they are more natively suited to running on the cloud. Our aim for thesis is to investigate suitable data storage for cloud. Considering the 'Big Data' scenario of today's world, we set forth to choose the NOSQL database model as the preferred solution for cloud computing. This paper aims to show two investigations on different branches of cloud data storage. The first analysis is based on the case study of performance benchmarking on 3 popular NOSQL databases - MongoDB, Cassandra, and HBase. The next part of investigation includes an experiment on the most popular 'Big Data' management framework – namely, Hadoop. Hadoop uses MapReduce for parallel computation, but writing MapReduce function is hard for programmers. So, our experiment is to configure HIVE data warehousing system on the top of Hadoop as a wrapper, so that end users gets benefit of using a SQL-like language, which is known as 'HiveQL' and provided by HIVE even if with the environment of complex MapReduce function.

Contents

Chapter 1

Introduction

The revolutionary prospect of cloud computing is changing the way of people's thought in IT. Day by day, the amount of data stored at companies like Google, Yahoo, Facebook, Amazon or Twitter has become incredibly huge. The new challenging requirement of this 'Big Data' era make us realize to rethink what we require of a database, and to come up with answers aside from the relational databases that have served us well for a quarter of century. Thus, web applications and databases in cloud are undergoing major architectural changes to take advantage of the scalability provided by the cloud.

1.1 Background

Only in the last century, data size would have been measured as 'Gigabytes to Terabytes'. This 'traditional data' had been well-managed by popular SQL database (RDBMS – Relational Database Management System). But the scenario has changed dramatically with the advent of 'Cloud Computing'. The advent of 'Cloud Computing' technology has caused a fundamental change to the nature of data. Now, in the 20^{th} century, data size is measured as 'Petabytes to Exabytes' and even with 'Zettabytes'. One Zettabyte is counted as 10^{21} bytes [1]. So, it is a huge amount of data.

We present a statistic of recent data explosion to have an idea of 'Big Data' scenario in today's world. [2] [3]

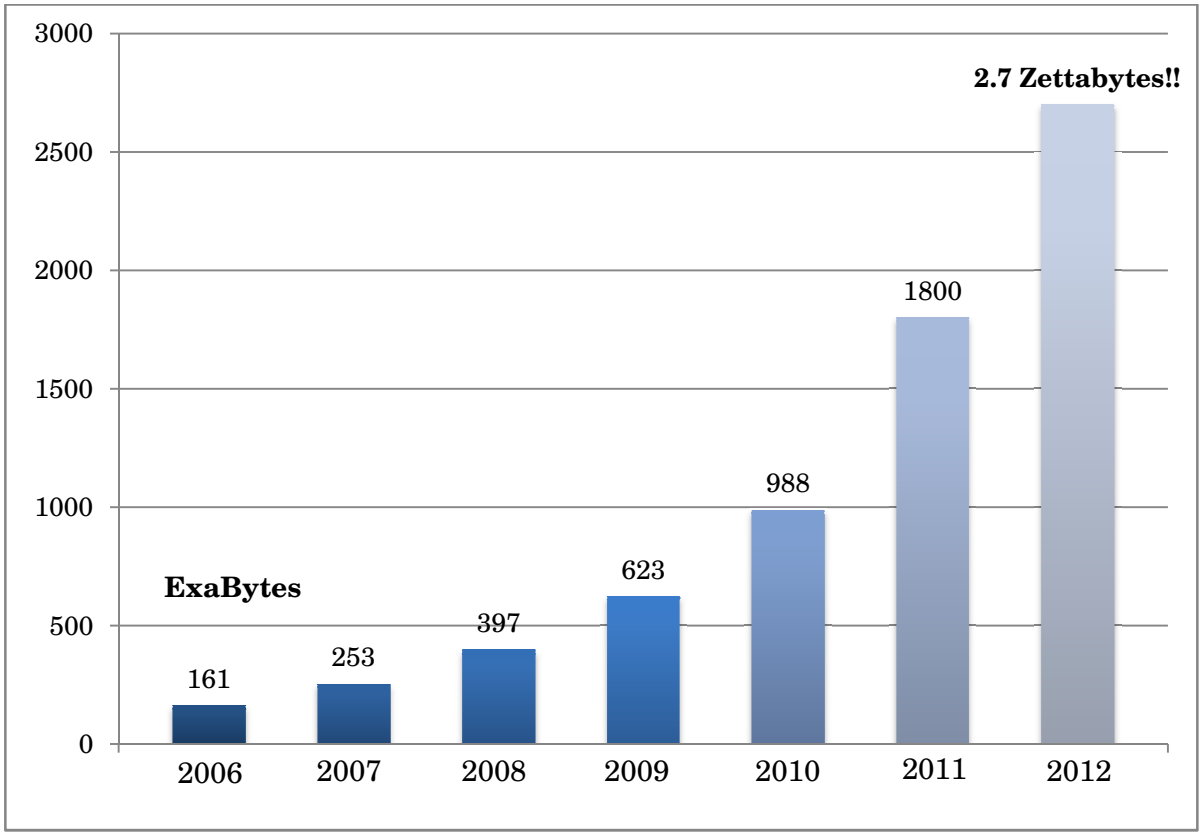


Figure 1.1 Recent Data Explosion

Also, to remember, data size is not the only issue to focus on. Instead of structured data, the variety of data types is increasing, namely unstructured text-based data and semi-structured data like social media data, location-based data, and log-file data. So, big web enterprises also need a 'Distributed database' instead of the 'Centralized database'.

1.2 Motivation

The background situation forces big web enterprises to think for a new database solution as traditional SQL database is not natively suited for cloud environment. A popular trend that is named as 'NOSQL' is emerging to solve the limits of SQL database. NOSQL breaks the one-eyed rule of relational database.

Also, we find new frameworks and analytic approaches are evolving rapidly. The most popular framework now-a-days is 'Hadoop'. Another special framework is 'Hive', which works as a wrapper on top of 'Hadoop'.

The evolving technologies motivates us to make research on back-end section of cloud computing.

In this paper, we aim to explore different branches of NOSQL database and make an experiment on 'Hadoop' and 'Hive' framework.

1.3 Thesis Outline

Chapter 2

This chapter analyzes different characteristics of two main types of cloud databases and conducts a comparison study to choose the better database tool.

Chapter 3

This chapter analyzes different categories of chosen database approach 'NOSQL' and presents a case study on performance benchmarking of 3 popular NOSQL databases, namely, MongoDB, Cassandra and HBase.

Chapter 4

This chapter investigates on the best knowing Data Management Framework, namely, Hadoop and its Programming Model – MapReduce.

Chapter 5

This chapter investigates on a Data Warehouse System – Hive, which is known to be used in Facebook which also solves the complex query procedure of Hadoop by using a SQL-like language that is named as HiveQL.

Chapter 6

This chapter summarizes our thesis work and gives an idea about our future plan.

Appendix

Appendix points out the implementation and configuration work on Hadoop and Hive.

Chapter 2

Cloud Data Storage Models

In this chapter, we will identify the available cloud data storage models and analyze their characteristics to pick the appropriate database approach for 'Big Data' evolution.

2.1 Available Cloud Data Storage Models

We set forth the approaches for cloud database to be counted as two main types –

1. SQL database model or RDBMS (Relational Database Management System)
2. NOSQL database model.

The traditional database model is SQL-based. It is known as RDBMS which has been around for more than 40 years and invented in 1970 by IBMer Edgar Codd. The main property of SQL database is that it uses relational algebra.

The second option for database choice is NOSQL. The acronym 'NoSQL' was first coined in 1998 by Carlo Strozzi [4]. NOSQL does not mean "No SQL"; it rather means "Not Only SQL". And the SQL word represents the relational databases, not the SQL language [5]. The idea for emerging this database is that both technologies can coexist and each has its place.

In the next section, we present characteristics of both databases.

2.2 Properties of SQL Database

SQL database has 3 major characteristics –

1. Fixed Schema
2. Relational Algebra
3. Query Language – SQL

In the following discussion, we briefly explain each of these properties.

2.2.1 Fixed Schema

SQL database follows a fixed schema condition. The term 'Fixed Schema' means - every requirements of database model have to be predefined [6].

2.2.2 Relational Algebra

As we have stated before, this the most import property of SQL database. The Relational Database Model states that - All information must be held in the form of a table. A table describes a specific entity type, and all attributes of a specific record are listed under an entity type. Each individual record is represented as a row, and an attribute as a column. Relations are represented as tables in the database through JOIN operation.

2.2.3 Query Language – SQL

The SQL database uses SQL as query language. SQL states for – Structured Query Language.

Example of SQL databases are: MySQL, Oracle, Microsoft SQL Server, PostgreSQL, IBM etc.

2.3 Properties of NOSQL Database

NOSQL database is quite different from SQL database in some significant ways. We find 4 major characteristics of NOSQL database –

1. Flexible Schema
2. Non-Relational Database
3. Simple Key-Value Stores

In the following discussion, we briefly explain each of these properties.

2.3.1 Flexible Schema

Flexible schema means - the schema can be changed according to the need for design and is defined by the program or data itself. So, conditions need not to predefine. NoSQL database systems are developed to manage large volumes of data. It follows the 'Flexible Schema' property.

2.3.2 Non-Relational Database

NOSQL is known as 'Non-Relational' database. Here, the term -'Non-Relational' does not mean "has no relations" or "cannot be described in terms of relational algebra." It means - "is not based on Edger Codd's relational database model". [7]

What a non-relational database **does not do** is - organize its data in related tables [8]. It does not have any 'JOIN' operations or constraints (i.e. NOT NULL) and does not require having 'Normalizing' format.

We present an example of 'Non-Relational Database' to the contrary of 'Relational Database' –
In a SQL database, a blog might have one table that stores posts and another table

that stores comments. A JOIN is then required to pull out all the comments along with a particular post. Each time, the relational database needs to define the relation through JOIN and other constraints.

On the other hand, NOSQL database does not require defining relations through constraints. With a non-relational database, one "collection" (the non-relational version of a *table*) would store all of the posts. Each comment associated with a post would be stored as part of that post's record within the collection. This means that one record (or "document", in non-relational terms) might contain just the post and no comments, another record might contain a much longer post and hundreds of comments. The benefit is that when we go to retrieve an individual post, we are automatically retrieving all the associated information (e.g., the comments for that post).

2.3.3 Simple Key-Value Stores

NOSQL database is simple Key-Value Stores. It makes the data retrieving more efficient. The Key-Value Store idea is more like 'Array Indexing'. For example, in a web service, a name is just a key and the whole data can be retrieved according to the name.

Example of NOSQL databases are: MongoDB, Cassandra, HBase etc.

So, here we summarized the properties of SQL and NOSQL database.

Now, in our next section, we conduct a comparison study on both database models to find the better suited database approach for cloud computing.

2.4 A Comparison Study – SQL vs. NOSQL

This section analyzes the issues on 'SQL vs. NOSQL Debate'. We consider 4 issues –

1. Schema
2. ACID vs. BASE Property
3. CAP Theorem
4. Scalability

In the following sub-sections, we made comparison on each issue.

2.4.1 1st Issue – Schema

SQL database follows 'Fixed Schema'. On the other hand, NOSQL database follows 'Flexible Schema'.

We find NOSQL database as the preferred solution for cloud computing. Our reasons for supporting NOSQL database are given below –

1. **Huge Data Size**
   This is the era of 'Big Data' where size of data is changing rapidly. We can we can think of Twitter as example. When it started out, it just collected bare-bones information with each tweet: the tweet itself, the Twitter handle, a timestamp, and a few other bits. Over its five-year history, though, lots of metadata has been added. A tweet may be 140 characters at most, but a couple KB is actually sent to the server, and all of this is saved in the database [9]. So, preparing a huge and fixed schema is quite impractical in such case.

2. **Continuously Changing Data Type**
   Not only the data size, but also the changing nature of data was our

consideration. Modern applications frequently deal with unstructured data: blog posts, web pages, voice transcripts, and other data objects that are essentially text. It is impossible to predict how data will be used, or what additional data these applications will need - as the project unfolds. For example, many applications are now annotating their data with geographic information: latitudes and longitudes, addresses. That almost certainly wasn't part of the initial data design. So, all these requirements cannot be predefined and thus, flexible schema is suitable in this case.

NOSQL has flexible schema as schema can be changed according to the need for design and is defined by the program or data itself. So, NOSQL is the better choice as database model in this case.

2.4.2 2nd Issue – ACID vs. BASE Property

SQL database has ACID Property. On the other hand, NOSQL database has the BASE property.

First, we describe each property here.

ACID Property

ACID stands for **A**tomicity, **C**onsistency, **I**solation and **D**urability. This property says that database transactions should be –

- **A**tomic: Everything in a transaction succeeds or the entire transaction is rolled back. [10]
- **C**onsistent: A transaction cannot leave the database in an inconsistent state.
- **I**solated: Transactions cannot interfere with each other.
- **D**urable: Completed transactions persist, even when servers restart etc.

BASE Property

BASE stands for **B**asically **A**vailable, **S**oft State and **E**ventual Consistency.

- **B**asic **A**vailability:

  This constraint states that the system does guarantee the availability of the data; there will be a response to any request. But, that response could still be 'failure' to obtain the requested data or the data may be in an inconsistent or changing state, much like waiting for a check to clear in anyone's bank account. [11]

- **S**oft-state:

  The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

- **E**ventual consistency:

  The system will *eventually* become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

The above description summarizes that - ACID compromises with 'Availability' for the sake of 'Consistency'. To the contrary, BASE compromises with 'Consistency' for the sake of 'Availability'.

Now, it depends on the type of application that which property we should give priority. Here, we are considering cloud computing environment where modern applications mostly need 'Availability' even if have to compromise with 'Consistency'. We state an example for better understanding of the scenario –

> Let's consider, we run an online book store and proudly display how many of each book we have in your inventory. Every time someone is in the process of

> buying a book, we lock part of the database until they finish so that all visitors around the world will see accurate inventory numbers. That works well if we run a small shop but not to run Amazon.com.
>
> Amazon might instead use cached data. Users would not see not the inventory count at this second, but what it was say an hour ago when the last snapshot was taken. Also, Amazon might violate the "I" in ACID by tolerating a small probability that simultaneous transactions could interfere with each other. For example, two customers might both believe that they just purchased the last copy of a certain book. The company might risk having to apologize to one of the two customers (and maybe compensate them with a gift card) rather than slowing down their site and irritating myriad other customers.

So, considering the cloud computing scenario, NOSQL is better over SQL again.

A question can arise here that "Why can't we have both 'Consistency' and 'Availability' at the same time?" We explain the answer in the next section through CAP theorem.

2.4.3 3rd Issue – CAP Theorem

CAP theorem was first coined by Eric Brewer in the year 2000 [12]. CAP stands for –

- **C**onsistency:

  All nodes see the same data at the same time.
- **A**vailability:

  Guarantee that every request receives a response about whether it was successful or failed.
- **P**artition Tolerance:

  The system continues to operate despite arbitrary message loss or failure of part of the system. So, operations will complete, even if individual components are unavailable.

The theorem states that -
"A distributed system *cannot ensure all three of the following properties at once. Web services can pick at most 2 out of these 3 requirements at a time."*

So, there are 3 options to choose for web services –

1. CA – Consistency & Availability
2. CP – Consistency & Partition Tolerance
3. AP –Availability & Partition Tolerance

Here, we analyze each scenario by giving example [13] –

1. **Scenario – 1: CA (Sacrificing Partition Tolerance)**

   On each of the three nodes, we will only store a subset of the user profiles. This is called sharding. Node one will have users A-H, node two I-S, and node three T-Z. As long as each node is up and running, we have achieved a three

times higher throughput than with a single node as each node only server a third of the traffic (assuming of course that user profile querying and updating is uniformly distributed through the alphabet). Consistency is achieved because immediately after data is written, it is accessible. Availability is achieved because each server is accessible in real time. However, we have lost the concept of partition tolerance as the disabling of one server has rendered a certain section of users unreachable. This carries the notion that upon hardware failure, data could have permanently been lost. All in all, not a good sacrifice under cloud environment.

2. **Scenario – 2: CP (Sacrificing Availability)**

   On each of the three nodes, we will store all the user profiles. And furthermore, to guarantee data consistency and data loss prevention, we will ensure that every write into the system happens on all three nodes before it is completed. So, if were to update a profile for Bob McBob, any subsequent queries or writes on Bob McBob's profile would be blocked until the update has completed. Even worse is when one of the nodes is lost but the requirement of three writes is still required, our entire system is unavailable until it is restored. This means that while our data is consistent and protected, we have sacrificed the availability of the data. This can be a reasonable sacrifice for cloud environment.

3. **Scenario – 3: AP (Sacrificing Consistency)**

   On each of the three nodes, we will store all the user profiles. However (and different than scenario B), we will acknowledge a completed write immediately and not wait for the other two nodes. This means that if a read comes in on node two for data written on node one, it may or may not be up-to-date depending on the latency of replication. We are still highly available

and still partition tolerant (with respect to the latency it takes to replicate to another second node). This is also a satisfying scenario under cloud environment.

Now, our aim was to select either SQL database or NOSQL database. We find that SQL database picks 'CA' property following CAP theorem. So, it sacrifices the most important property for a 'Distributed Database' that is – 'Partition Tolerance'. On the other hand, NOSQL database sacrifices either 'Consistency' or 'Availability' and picks between 'CP' and 'AP'.

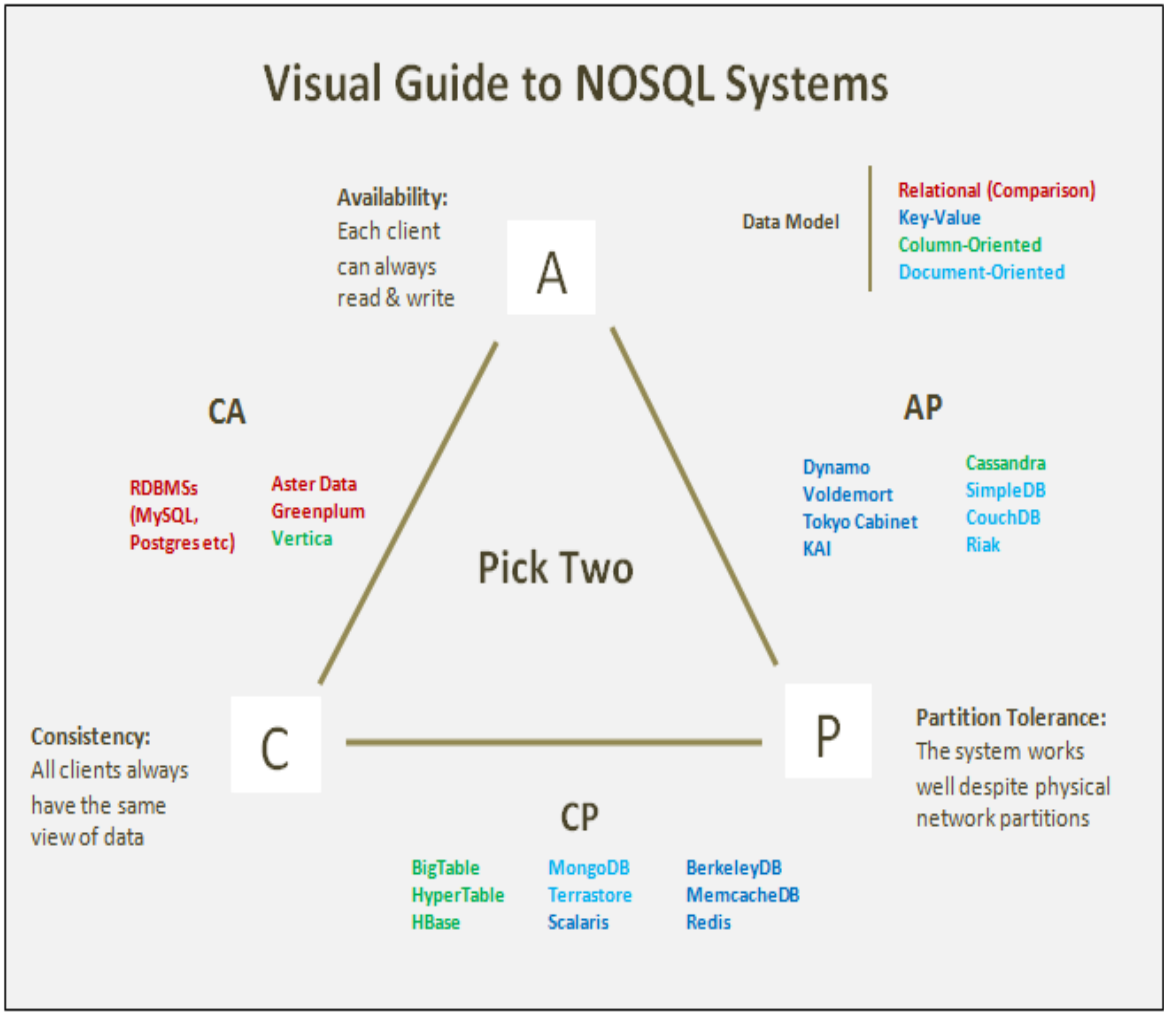


Figure 2.1 CAP Theorem

Cloud computing technology is built upon the idea of 'Distributed Database'. So, if 'Partition tolerance' is not ensured then cloud technology will not survive. So, 'CA' is not a preferred choice for web services and they are forced to choose between 'CP' and 'AP'.

That makes the conclusion that according to CAP theorem; again, NOSQL database wins over SQL database.

2.4.4 4th Issue – Scalability

Scalability means the capability to cope and perform under an increased or expanding workload.

A system that scales well will be able to maintain or even increase its level of performance or efficiency when tested by larger operational demands [14]. That is one of the fundamental requirement of cloud computing. So, 'Scalability' is considered as a major issue while choosing cloud database.

We can have 2 types of scalability –

1. **Horizontal Scalability or Scale Out**
   Horizontal Scalability means adding more individual units of resource doing the same job (add an extra node to the cluster).
2. **Vertical Scalability or Scale Up**
   Vertical Scalability means taking a single unit of resource (i.e. RAM) and making it larger.

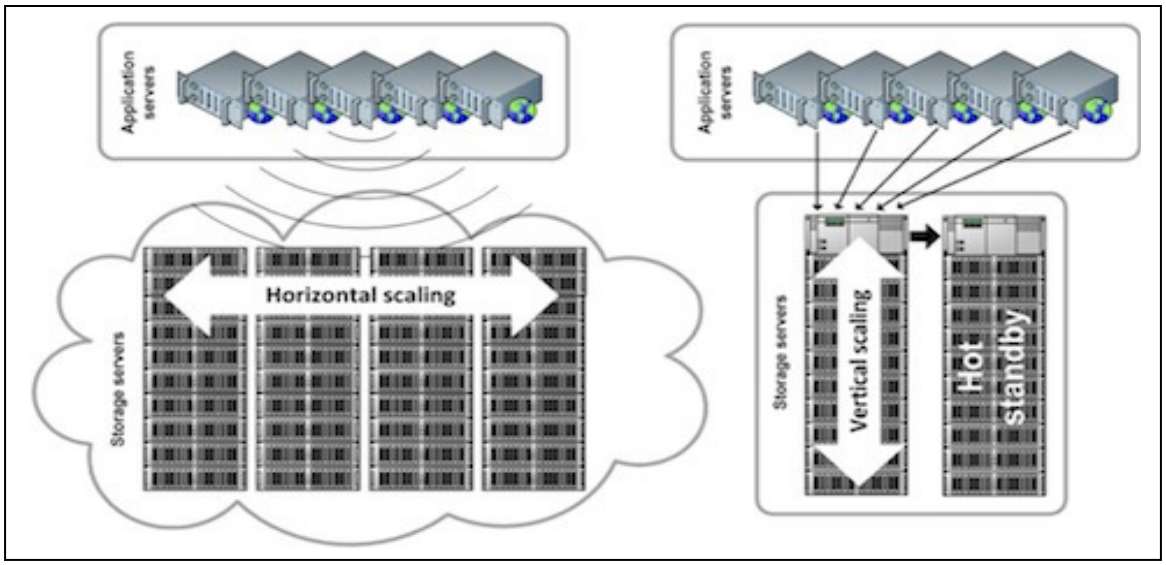


Figure 2.2 Horizontal Scalability vs. Vertical Scalability

'Horizontal Scalability' is said to be the better scalability option as we can scale indefinitely. On the other hand, 'Vertical Scalability' always runs into limits as increasing performance of a single node server has a finite level.

Now, to pick the right database, we analyze the 'Horizontal Scalability' performance between SQL and NOSQL database. We find that the fundamental option to gain 'horizontal scalability' in a distributed system is – 'Sharding'. 'Database Sharding' can be simply defined as a 'shared-nothing' partitioning scheme. If we think of broken glass, we can get the concept of sharding - breaking our database down into smaller chunks called 'shards' and spreading those across a number of distributed servers. Sharding can be achieved in 2 ways –

1. **Sharding Manually:** SQL database shard manually.
2. **Sharding Automatically:** NOSQL database shard automatically.

Between the 2 types of sharding, it is found obvious that 'Automatic Sharding' is preferred for a distributed system. But –

1. SQL database cannot shard automatically because of its table-based nature. In SQL, multiple tables may be locked for modification during transaction. If those tables are spread across multiple shards/servers, it'll take more time to acquire the appropriate locks, update the data and release the locks. So, scalability is not well achieved in SQL database.
2. To the contrary, a NOSQL database shard automatically as this database does not distribute a logical entity across multiple tables; it's always stored in one place. They do not enforce referential integrity between these logical entities. They only enforce consistency inside a single entity and sometimes not even that.

Here, we present an example to show the scenario how 'Automatic Sharding' enables NOSQL database to scale in a better way.

If we were to write 20 entities to a database cluster with 3 nodes [15] –

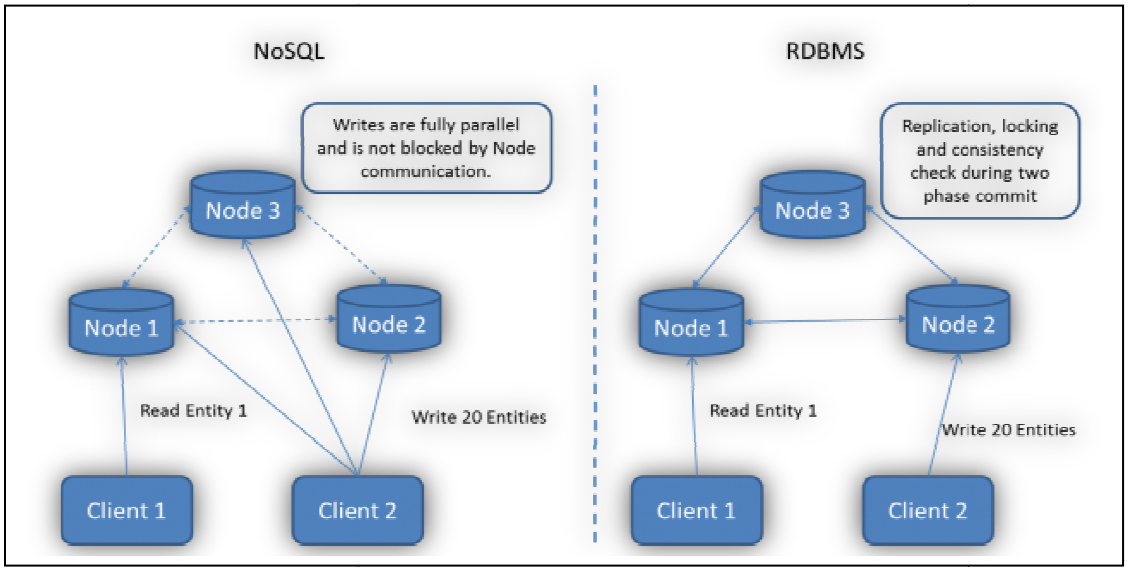


Figure 2.3 NOSQL Performance Compared to SQL Performance in the context of 'Relational Property' of SQL Database

In NoSQL

- We can write independently on all three nodes **because** the database does not need to synchronize between the nodes.
- Client 1 might see changes on Node 1 before Client 2 has written all 20 entities **because** there's no need for a two-phase commit.

In SQL

A distributed RDBMS solution on the other hand needs to enforce ACID consistency across all three nodes:

- RDBMS needs to read data from other nodes in order to ensure referential integrity **because** of synchronization.
- **Until** all three nodes acknowledged a two phase commit, Client 1 will either not see any or will be blocked until that happened.

All these happens during the transaction and blocks Client 2.

So, it can be concluded that though SQL database can have manual sharding but it does not give good performance because of its table-based nature. On the other hand, NOSQL does not follow 'Relational' concept, so, it can provide better performance. So, NOSQL is our preferred choice under the issue of 'Scalability'.

2.5 Chosen Database Approach – NOSQL

For all the above 4 issues of 'Schema', 'ACID vs. BASE Property', 'CAP Theorem' and 'Scalability' we find that NOSQL wins in all cases.

Also, we present a table to show the list of popular big web sites that use NOSQL database as their database approach.

NOSQL Database	Popular Companies that are using NOSQL Database
Cassandra	Digg, Facebook, Twitter, Redit [16] [17]
MongoDB	Foursquare, The New York Times [18]
HBase	Facebook [19]
DynamoDB	Amazon
BigTable	Google
CouchDB	CERN, BBC, Interactive Mediums
Voldemort	LinkedIn
Redis	Facebook, Digg, GitHub [20]
Riak	Widescript, Western Communication

Table 2.1: List of sites that are using NOSQL database. The above table is the mirror reflection of importance of NOSQL database in cloud computing.

Most of the big websites have moved to NOSQL database. So, we select NOSQL as our preferred database approach for cloud computing and aim to make further investigation on NOSQL.

Chapter 3

The NOSQL Movement

The comparison study on 'SQL vs. NOSQL' leads us to enter into a new world of NOSQL database – a world that is built with non-relational concept. In this chapter, we aim to explore different branches of NOSQL database and show a performance analysis on 3 popular NOSQL databases in the context of cloud computing needs.

3.1 Classification of NOSQL Database Models

To understand the vast arena of NOSQL database concept, we first go through the possible categories of this database model.

NOSQL data stores can be classified into four [21] main categories:

1. Key-value Stores
2. Column Family Stores
3. Document Databases
4. Graph Databases

In the following sub-sections, each category is briefly described in the order of database model concept, example application, available NOSQL databases, strengths and weaknesses.

3.1.1 Key-value Stores

Key-Value stores are considered as the most ubiquitous technology under the NoSQL banner. The main idea here is using a hash table where there is a unique key and a pointer to a particular item of data. [22]

(a)

id	name	mood	birth_date	color
12	Stella	Happy	2007-04-01	NULL
13	Wimma	Hungry	NULL	black
9	Ninja	NULL	NULL	NULL

(b)

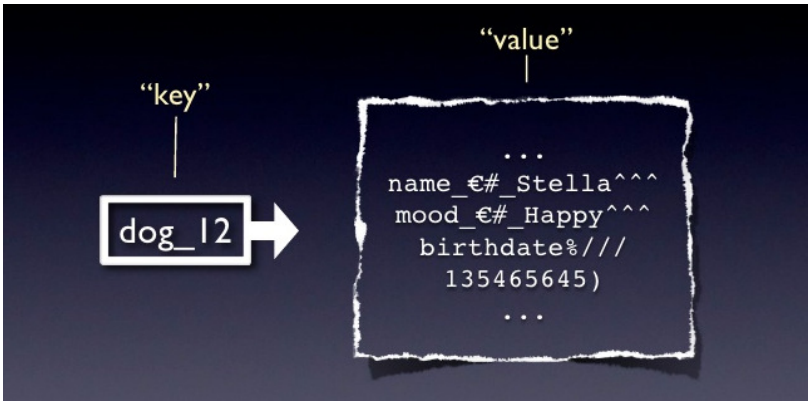


Figure 3.1 Example of Key-value Store model. In (a), data is stored in a table with row-column property in SQL database. In (b), each value has a unique key to store and retrieve data in Key-value Store database.

Typical Applications	Content caching (Focus on scaling to huge amounts of data, designed to handle massive load), logging, etc.
Database Examples	DynamoDB, Redis, Voldemort.
Strengths	Fast lookups
Weaknesses	Stored data has no schema

Table 3.1 Key-value Stores Use Case

3.1.2 Document Databases

Document databases are essentially the next level of Key/value. The model is basically versioned documents that are collections of other key-value collections. The semi-structured documents are stored in formats like JSON. Document database allows nested values associated with each key that is not done in 'Key-value Stores' case. For that advantage, Document databases support querying more efficiently.

(a)

id	name	mood	birth_date	color
12	Stella	Happy	2007-04-01	NULL
13	Wimma	Hungry	NULL	black
9	Ninja	NULL	NULL	NULL

(b)

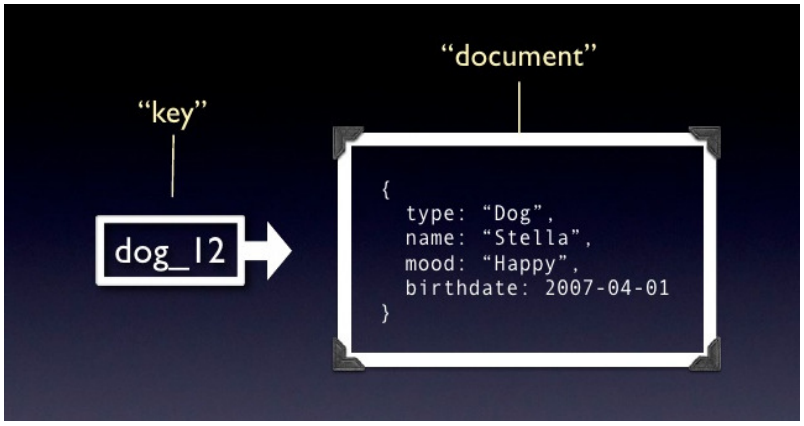


Figure 3.2 Example of Document Databases model. In (a), data is stored in a table with row-column property in SQL database. In (b), each unique key stores the whole document with nested collections of key-value.

Typical Applications	Web applications (Similar to Key-Value stores, but the DB knows what the Value is)
Database Examples	CouchDB, MongoDB
Strengths	Tolerant of incomplete data [23]
Weaknesses	No standard query syntax

Table 3.2 Document Databases Use Case

3.1.3 Column Family Stores

Column Family Stores were created to store and process very large amounts of data distributed over many machines. Like the concept of 'Key-value Stores', there are still keys but they point to multiple columns. Then, the columns are arranged by column family.

(a)

id	name	mood	birth_date	color
12	Stella	Happy	2007-04-01	NULL
13	Wimma	Hungry	NULL	black
9	Ninja	NULL	NULL	NULL

(b)

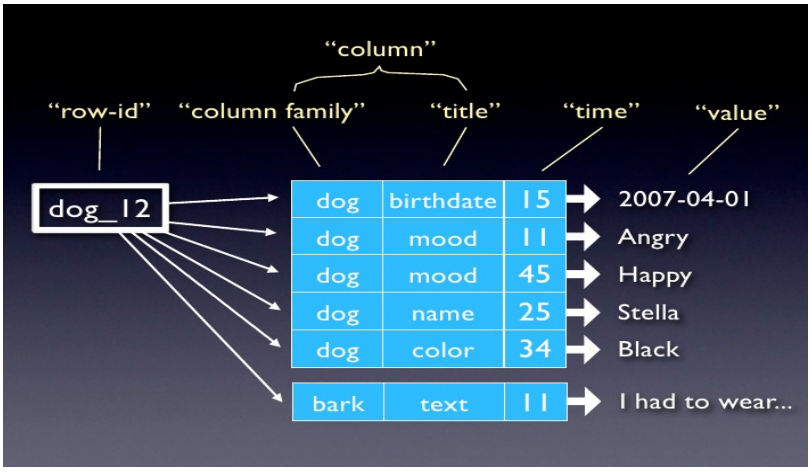


Figure 3.3 Example of Column Family Stores model. In (a), data is stored in a table with row-column property in SQL database. In (b), each unique key points to multiple columns.

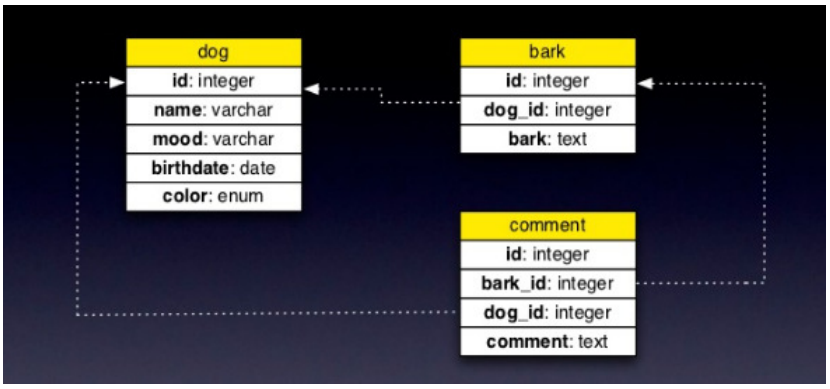
Typical Applications	Distributed file systems
Database Examples	Cassandra, HBase, BigTable, HyperTable
Strengths	Fast lookups, good distributed storage of data
Weaknesses	Very low-level API

Table 3.3 Column Family Stores Use Case

3.1.4 Graph Databases

"Relational database is a collection loosely connected tables" whereas "Graph is a multi-relational graph." The main drawback of SQL database is that – each time we have to define relationship between tables by using constraints because of its rigid structure of tables and row-columns. So, relationships are weak in SQL database. Graph databases solve the problem as relationships are first class citizen for these databases. In this category of NOSQL database, a flexible graph model is used which, again, can scale across multiple machines and can map data using relations.

(a)



(b)

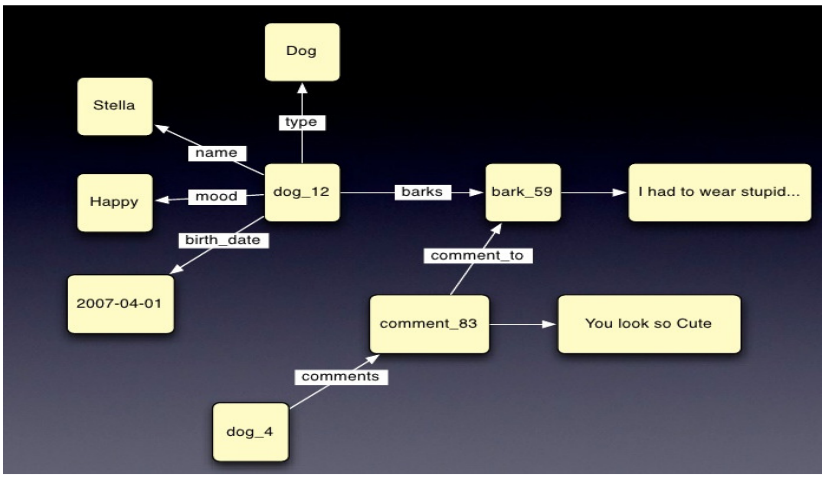


Figure 3.4 Example of Graph Databases model. In (a), a schema for **Figure 3.1 (a)** is drawn to show the rigid relationship between tables in SQL database. In (b), a Graph database model is drawn to show the flexibility of relationships. [24]

Typical Applications	Social networking, Recommendations (Focus on modeling the structure of data – interconnectivity)
Database Examples	Neo4J, InfoGrid, Infinite Graph
Strengths	Graph algorithms e.g. shortest path, connectedness, n degree relationships, etc.
Weaknesses	Has to traverse the entire graph to achieve a definitive answer. Not easy to cluster.

Table 3.4 Graph Databases Use Case

3.2 A Case of Study: Evaluating NOSQL Performance using YCSB Benchmark Results

In the previous section, we presented a brief introduction on NOSQL database. The immense field of NOSQL technology, coupled with a lack of apples-to-apples performance comparisons, makes it difficult to understand the tradeoffs between systems and the workloads for which they are suited. So, in our case study we aim to measure performance of some selected NoSQL products and to determine the best use cases of each product for different internet services.

We selected 3 NOSQL databases to conduct study by recollecting popular CAP theorem image by Eric Brewer.

	NOSQL Database	CAP Property	Data Model
1	MongoDB	CP (Consistency & Partition Tolerance)	Document Databases
2	Cassandra	AP (Availability & Partition Tolerance)	Column Family Stores
3	HBase	CP (Consistency & Partition Tolerance)	Column Family Stores

Table 3.5 Selected NOSQL Databases for Benchmarking

3.2.1 Test Framework: YCSB

To analyze performance of our selected NOSQL databases, we choose to consider the benchmark test results of "Yahoo! Cloud Serving Benchmark" (YCSB) framework. The tool was first invented by 'Yahoo! in the year 2010 [25]. This tool allows benchmarking multiple systems and comparing them by creating "work loads". Using this tool, one can install multiple systems on the same hardware configuration, and run the same workloads against each system. Then it is possible to plot the performance of each system (for example, as latency versus throughput curves) to see when one system does better than another. [26]
YCSB currently supports - Cassandra, HBase, MongoDB, Voldemort and JDBC.

In this section, we give an overview on YCSB work procedure.

YCSB Architecture

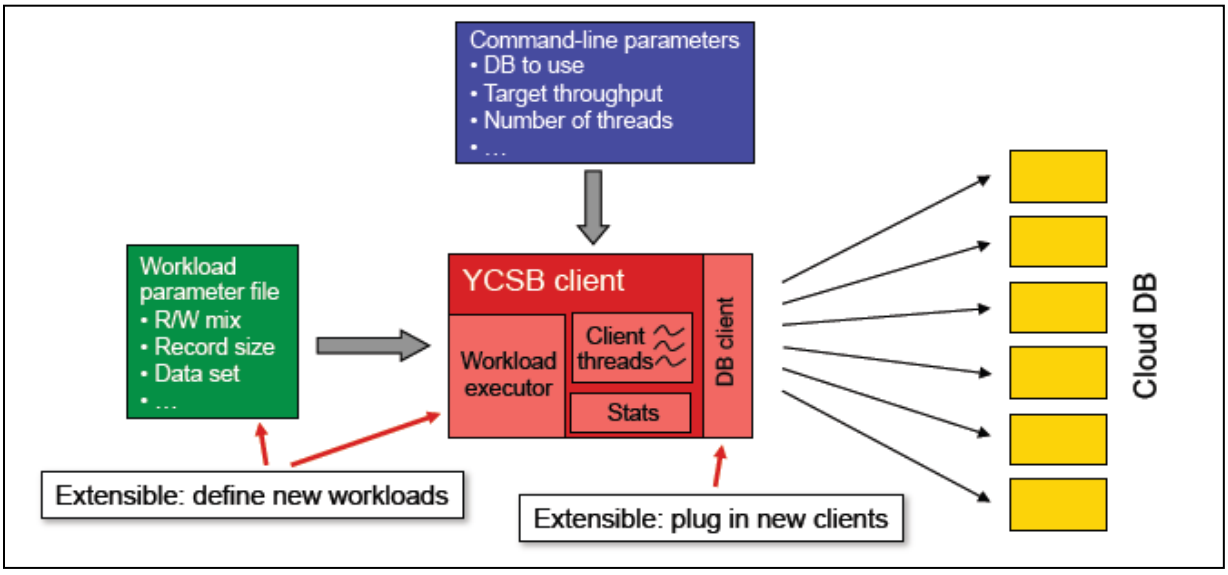


Figure 3.5 YCSB Architecture

The YCSB Client is a Java program for generating the data to be loaded to the database, and generating the operations which make up the workload.
YCSB has four types of operations–

1. Insert
2. Update
3. Read and
4. Scan.

The architecture of the client is shown in **Figure 3.5**.

DB Interface Layer

YCSB Client uses the DB interface layer to send commands to the configured database i.e. Cassandra.

Workload Executor

The Workload defines the data that can be loaded and executed in two executable phases:

1. The **Loading** phase, which defines the data to be inserted and
2. The **Transactions** phase, which defines the operations to be executed against the data set. [27]

Interaction between Workload Executor and DB Interface Layer

1. Workload executor drives multiple client threads.
2. Each thread executes a sequential series of operations by making calls to the database interface layer, both to load the database (the load phase) and to execute the workload (the transaction phase).
3. At the end of the experiment, the statistics module aggregates the measurements and reports average, 95th and 99th percentile latencies, and either a histogram or time series of the latencies.

3.2.2 YCSB Benchmark Results

The YCSB Benchmark Test Case that we analyzed has some system specifications.

Selected Versions of Databases

1. MongoDB-1.8.1
2. Cassandra-0.7.4
3. HBase-0.90.2

Test Cases

The benchmark is conducted on the basis of three test cases [28] –

1. Insert Only
2. Read Only
3. Read & Update

Test Workload

The test workload is as follows.

Insert Only

Enter 50 million 1K-sized records to the empty DB.

Read Only

Search the key in the *Zipfian Distribution*[1] for a one hour period on the DB that contains 50 million 1K-sized records.

Read & Update

Conduct '**Read & Update'** one-on-one instead of '**Read'** under the identical conditions of 'Read Only'.

[1] The Zipf distribution, sometimes referred to as the zeta distribution, is a discrete distribution commonly used in linguistics, insurance, and the modeling of rare events.

Test Results

The benchmark test results are shown in the following figure.

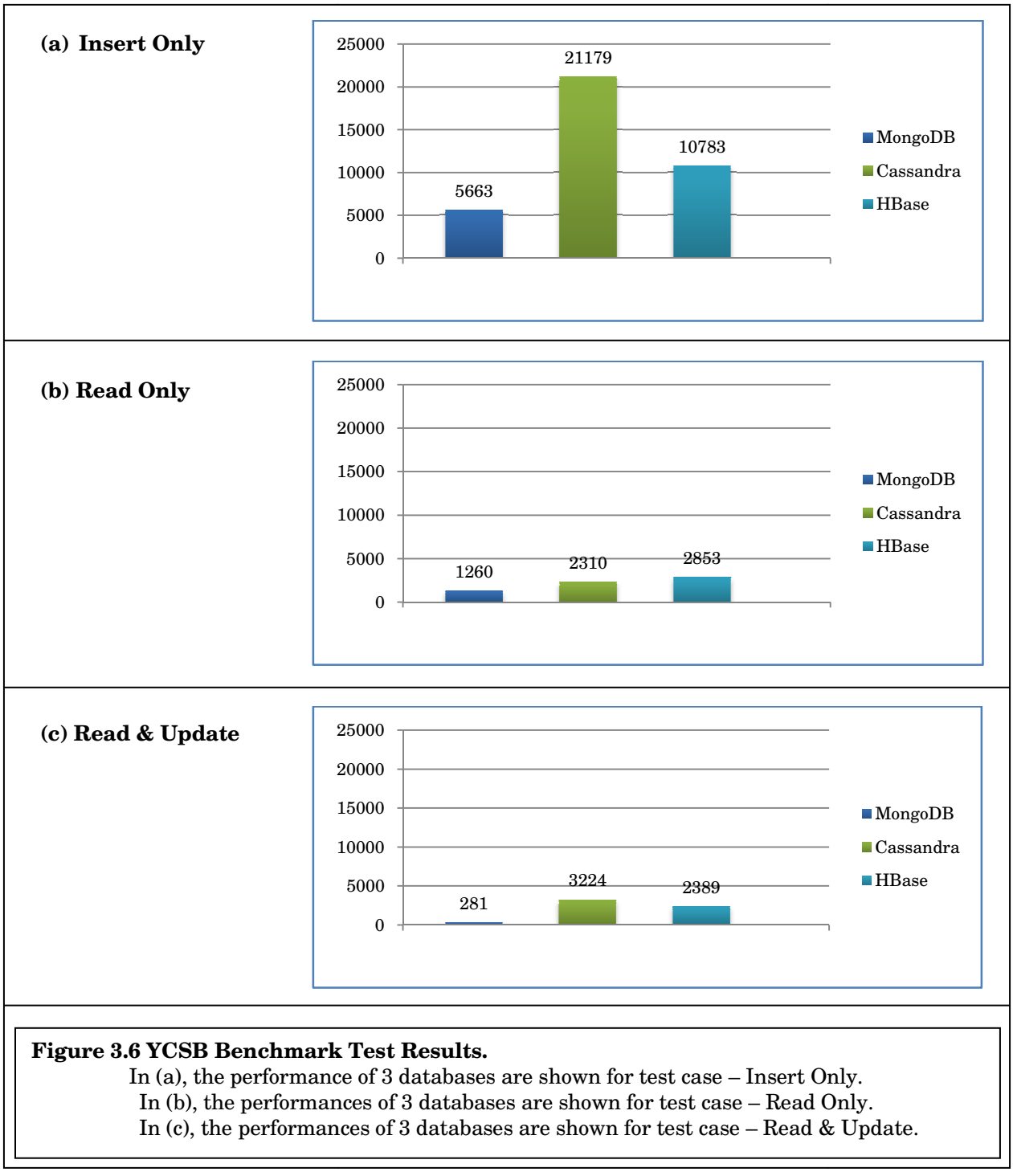


Figure 3.6 YCSB Benchmark Test Results.
In (a), the performance of 3 databases are shown for test case – Insert Only.
In (b), the performances of 3 databases are shown for test case – Read Only.
In (c), the performances of 3 databases are shown for test case – Read & Update.

3.2.3 Summary of Benchmark Test Result

We summarize the benchmark test result in 3 test cases.

	Test Case	MongoDB Performance	Cassandra Performance	HBase Performance
1.	Insert Only	Satisfactory but lowest amongst 3 databases	Outstanding throughput	Relatively Good
2.	Read Only	Lowest amongst 3 databases	Relatively Good	Highest amongst 3 databases
3.	Read & Update	Lowest amongst 3 databases	Highest amongst 3 databases	Relatively Good

Table 3.6 Summary of YCSB Benchmark Test Result

The benchmark test concludes the following comparison results –

1. **'Read Only' and 'Read & Update' are much slower than 'Insert Only' operations in these NoSQL solutions.**
2. Cassandra's performance in 'Insert Only' and 'Read & Update' was better than the other two products. **But Cassandra is slower in reading than writing.**
3. **HBase's performance was better than Cassandra in 'Read Only'.** HBase also shows relatively good performance in 'Insert Only' and 'Read & Update'.
4. **MongoDB's throughput in all three conditions was the lowest of the three products.**

In the next section, we examine the reason behind the performance variation of each database.

3.3 Benchmark Result Analysis

The benchmark test in the previous section gave different performance result for the three databases and we summarized the result with 4 conclusions. In this section, we aim to analyze each result case in the context of Insert/Write and Read operations for the 3 selected databases – Cassandra, HBase and MongoDB.

3.3.1 Result Case – 1:

'Read Only' and 'Read & Update' Operation are much Slower than 'Insert Only' operation

Both Cassandra and HBase follow Column Family Stores Data model. Internal architecture of these databases is based on Google's BigTable model, although, Cassandra was directly influenced by Amazon's Dynamo. So, we analyze this result case in the context of BigTable's internal structure.

We first present the data model of BigTable.

BigTable Data Model

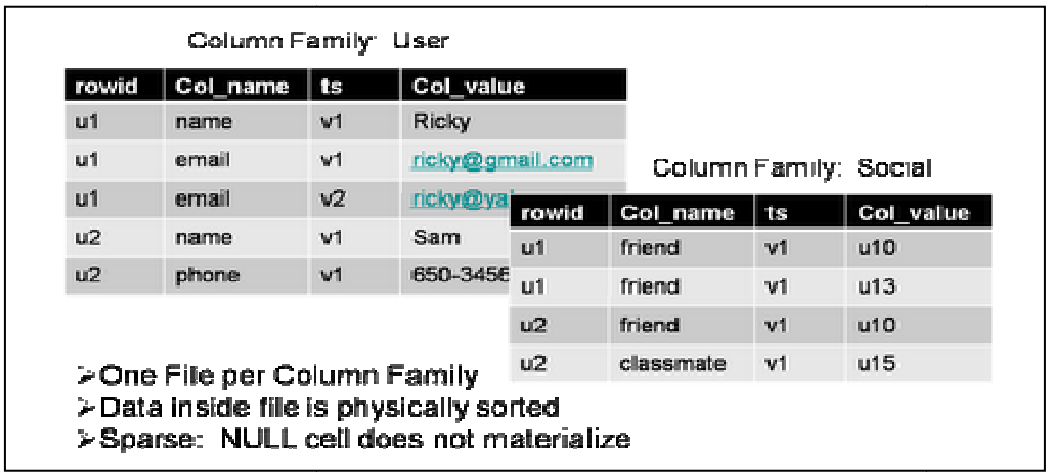


Figure 3.7 Column-Oriented Database Model of BigTable

BigTable is a Column-Oriented Database that stores data in a Multidimensional Sorted Map format and has the <row key, column family, column key> data structure.

Column family is a basic unit that stores data with column groups that are related to each other. BigTable store each *column family* contiguously on disk (i.e. one file per column family), and physically sort the order of data by *row id, column name and timestamp*. After that, the sorted data will be compressed so that a disk block size can store more data. On the other hand, since data within a column family usually has a similar pattern, data compression can be very effective.

Based on this data model, BigTable conducts its Write/Read operations. Now, from the architectural perspective, we identify the reason of performance difference in Write/Read operation for Cassandra and HBase.

Write Operation in BigTable

Figure 3.7 shows the internal structure of Write/Read path in BigTable –

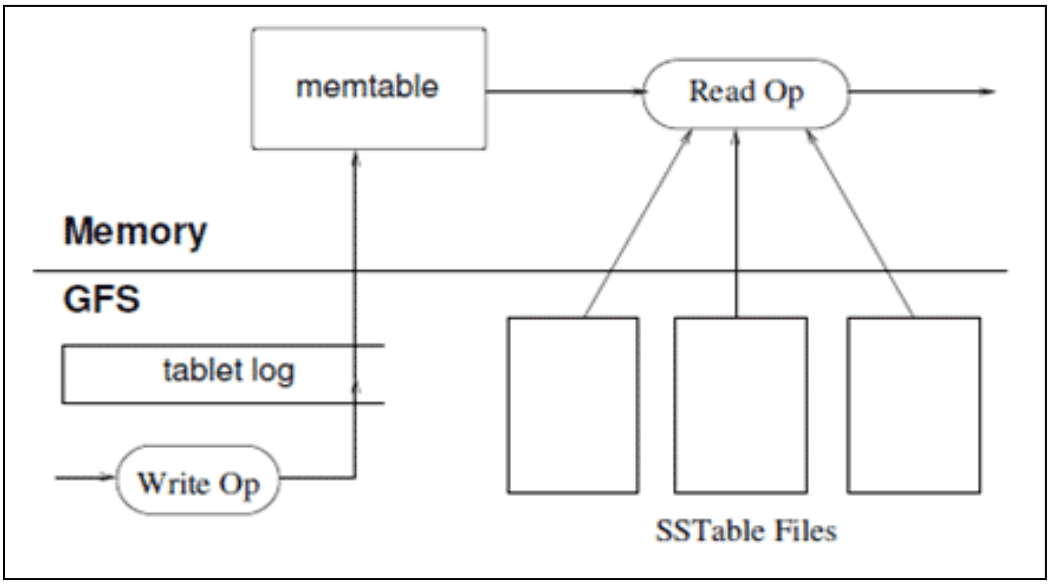


Figure 3.8 Data Read/Insert Path of Google's BigTable

BigTable model is highly optimized for write operation with sequential write. In BigTable, data is written basically with the append method. In other words, when modifying data, the updates are appended to a file, rather than an in-place update in the stored file.

Write operation is completed in 2 steps –

1. When a write operation is inserted, it is first placed in a memory space called ***memtable***. All the latest update therefore will be stored at the ***memtable*** at first.
2. If the ***memtable*** is full, then the whole data is stored in a file called ***SSTable*** *(**S**orted **S**tring* ***Table**)*. The table is sorted by String key. Over a period of time there will be multiple ***SSTables*** on the disk that store the data.

Read Operation in BigTable

Now, while doing 'Read' operation, an extra amount of time is needed because if data is not present in the memtable, we need to do **'Merge Read'** –

1. Whenever a read request is received, the systems will first lookup the Memtable by its 'row key' to see if it contains the data. [29]
2. If not, it will look at the on-disk SSTable to see if the row-key is there.

We call this the 'Merged read' as the system need to look at multiple places for the data. ***SSTable*** has a companion ***Bloom filter*** such that it can rapidly detect the absence of the row-key. In other words, only when the bloom filter returns positive will the system be doing a detail lookup within the SSTable.

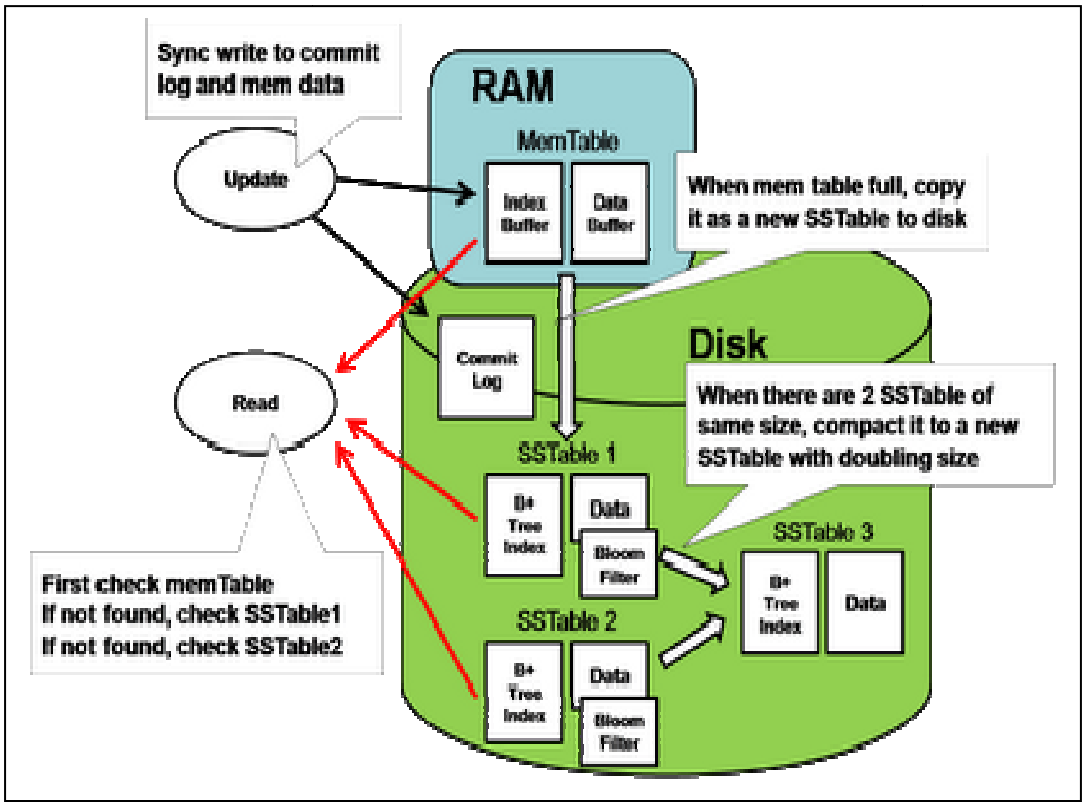


Figure 3.8 Read Operation in Google's BigTable

Another concept to improve read efficiency is – **'Periodic Data Compaction'.** As we can imagine, it can be quite inefficient for the read operation when there are too many ***SSTables*** scattering around. Therefore, the system periodically merges the ***SSTable***, since, each of the ***SSTable*** is individually sorted by key, a simple 'Merge sort' is sufficient to merge multiple ***SSTables*** into one. The merge mechanism is based on a logarithm property where two ***SSTable*** of the same size will be merging into a single ***SSTable***, will doubling the size. Therefore the number of ***SSTable*** is proportion to O(logN) where N is the number of rows.

So, we have discussed the issues behind 'Write' and 'Read' operation. In the case of 'Write' operation, at first, data is recorded in the memory and then, moved to the actual disk only after a certain amount has been accumulated. It thus improves the efficiency of 'Write' operation. On the other hand, the concept of 'Merge Read' and 'Periodic Data Compaction' requires extra time to complete 'Read' operation.

3.3.2 Result Case – 2:
Cassandra Performance – Faster in Writing than Reading

As we mentioned before, Cassandra additionally uses Amazon's Dynamo Model along with Google's BigTable model. Cassandra follows Dynamo's DHT (distributed hash table) model to partition its data. It is known as 'Consistent Hashing'. Through 'Consistent Hashing', each machine (node) is associated with a particular id that is distributed in a ***keyspace*** (e.g. 128 bit). The entire data element is also associated with a key (in the same key space). The server owns all the data whose key lies between its id and the preceding server's id.

Now, in this analysis, we state the steps that Cassandra goes through to complete 'Write' and 'Read' operation. Thus, difference between 'Write' and 'Read' performance will be revealed.

Write Operation in Cassandra

The steps during 'Write' operation in Cassandra is as follows –

1. Client submits its write request to a single, random Cassandra node. [30]
2. This node acts as a proxy and writes the data to the cluster where cluster of nodes is stored as a "ring" of nodes.
3. Now, using the **'Replication Placement Strategy'**, writes are replicated to N nodes. [31]
4. Finally, the node waits for the N successes and then returns success to the client.

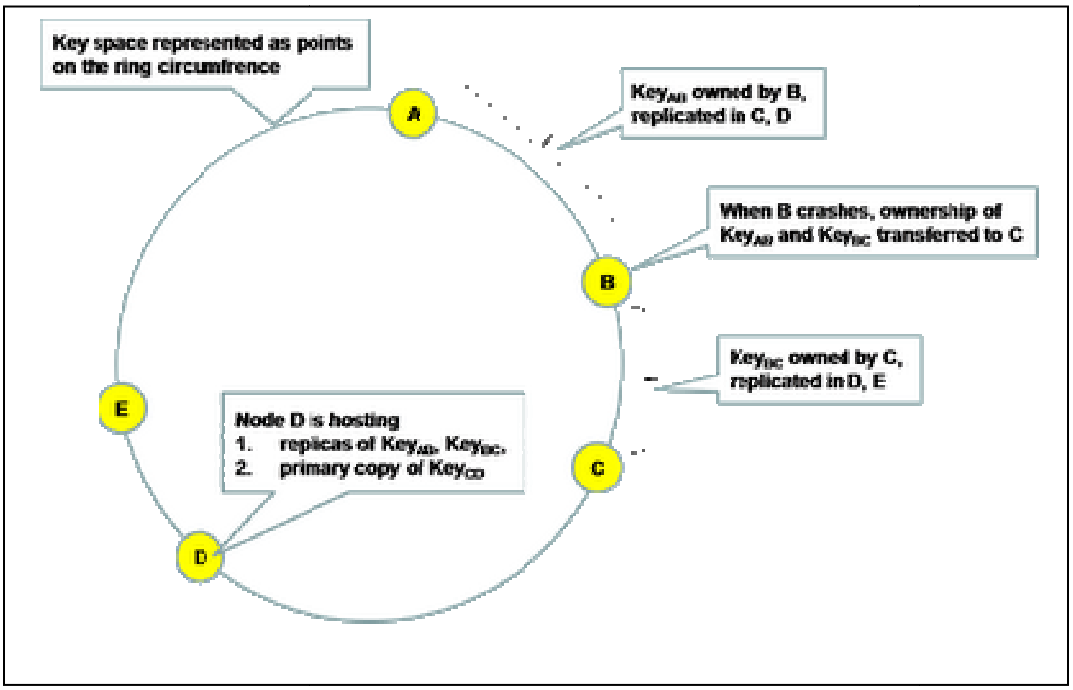


Figure 3.9 Simple Partition Strategies in Cassandra

In case any node is failed, the write operation can be retried at a later using '**Hinted handoff**'. According to this process, the failed operation will pick a random node as a handoff node and write the request with a hint telling it to forward the write request back to the failed node after it recovers. The handoff node will then periodically check for the recovery of the failed node and forward the write to it. Therefore, the original node will eventually receive the entire write request.

In this way, Cassandra performs a faster 'Write' operation that ensures 'Availability' (Figure 2.1: CAP Theorem).

Read Operation in Cassandra

The steps during 'Write' operation in Cassandra is as follows –

1. A client makes a read request to a random node. [32]
2. The node acts as a proxy determining the nodes having copies of data.
3. The node requests the corresponding data from each node.
4. Now, while returning data, Cassandra allows the client to select the strength of the read consistency –

   Single read: The proxy returns the first response it gets. This can easily return stale data. [33]

   Quorum read: The proxy waits for a majority to respond with the same value.
5. Finally, a value will be returned to client and thus, 'Read' operation is completed.

   In the background, the proxy also performs 'Read Repair' on any inconsistent responses. According to that method, when the client performs

a ‘Read’, the proxy node will issue N reads but only wait for R copies of responses and return the one with the latest version. In case some nodes respond with an older version, the proxy node will send the latest version to them asynchronously; hence these left-behind nodes will still eventually catch up with the latest version.

We state an example here. For example, we have a key “A” with a value of “123” in our cluster. Now we update “A” to be “456”. The write is sent to N different nodes, each of which takes some time to write the value. Now we ask for a read of “A”. Some of those nodes might still have “123” for the value while others have “456”. They will all eventually return “456”. This is also known as ‘Eventual Consistency’.

The situation in ‘Quorum Read’ makes it much more difficult to get stale data but this is the reason why ‘Read’ operation in Cassandra tends to be slower than ‘Write’ operation.

So, we have discussed the issues behind ‘Write’ and ‘Read’ operation. Since the success of replicated writing is not guaranteed, the data suitability is checked in the reading stage. That makes Cassandra to give slower performance in ‘Read’ operation than ‘Write’ operation.

3.3.3 Result Case – 3:

HBase Performance – Faster in Reading compared to Cassandra

HBase has the same structure as BigTable. Based on the BigTable, HBase uses the Hadoop Distributed File System (HDFS) as its data storage engine. The advantage of this approach is then HBase doesn't need to worry about data replication, data consistency and resiliency because HDFS has handled it already.

So far we have analyzed that Cassandra is faster in 'Write' operation than 'Read' operation. But **Figure 3.6(b)** shows that HBase has better performance in 'Read' operation than Cassandra. So, in this section, we aim to identify the reason for what HBase shows the faster performance in 'Reading' than Cassandra by analyzing both 'Write' and 'Read' paths of HBase Memstore.

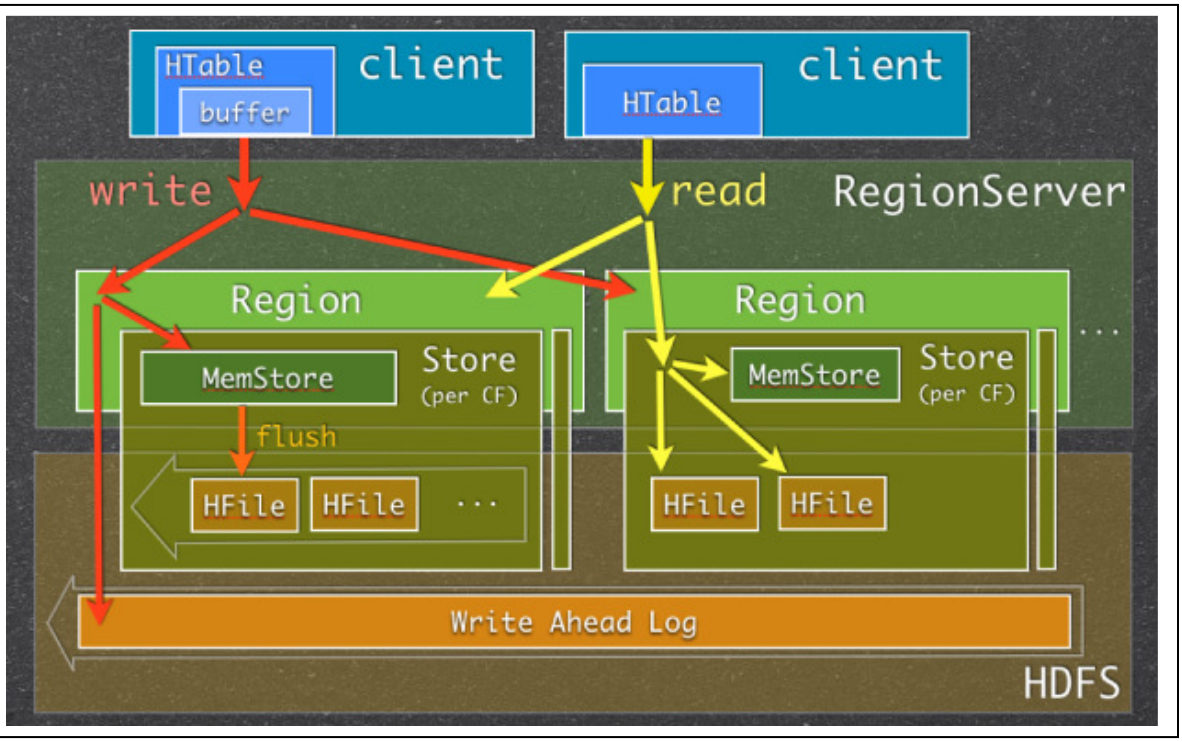


Figure 3.10 Memstore Usage in HBase Read/Write Paths

Write Operation in HBase

The 'Write' paths in HBase can be described as follows –

- **RegionServer (RS)** receives write request and it directs the request to specific **Region**. [34]
- Each **Region** stores set of rows. Rows data can be separated in multiple **Column Families** (CFs).
- Data of particular **CF** is stored in **HStore** which consists of **Memstore** and a set of **HFiles**.
- Memstore is kept in **RS** main memory, while **HFiles** are written to HDFS.
- When write request is processed, data is first written into the **Memstore**. Then, when certain thresholds are met (obviously, main memory is well-limited) **Memstore** data gets flushed into **HFile**.

Read Operation in HBase Memstore

The reading end things in HBase are simple –
HBase first checks if requested data is in **Memstore**, then goes to **HFiles** and returns merged result to the user.

The discussion concludes that HBase only 'writes' on a single region in the beginning, and receives requests on only one node. While 'reading', HBase only reads data once. On the other hand, Cassandra reads the data three times to check data suitability. So, 'Reading' performance of HBase is faster compared to Cassandra.

3.3.4 Result Case – 4:

MongoDB Performance – Lowest Throughput among the 3 Databases

The benchmark test of YCSB in **Section 3.2** shows that MongoDB has the lowest performance in all 3 cases among the 3 databases. We identify the reason behind the scenario in this section –

1. Unlike Cassandra and HBase, MongoDB does not follow 'Column Family Stores' data model. Rather, MongoDB uses a 'Document Database' model. According to this data model, each key is associated with a nested amount of values. So, memory size plays an important role in MongoDB. MongoDB **operates on a memory base and places high performance above data scalability**. If reading and writing is conducted within the usable memory, then high-performance is possible. However, performance is not guaranteed if operations exceed the given memory. That is the reason why MongoDB shows poor performance in all 3 cases.
2. However, the MongoDB has been found to record greater performance than Cassandra or HBase, if 300 thousand records are taken instead of 50 million as workload.

So, MongoDB can be used quickly, schema-free when using a certain amount of data.

To conclude the performance result, each NOSQL database has its distinct functionalities. MongoDB is used in Foursquare, SourceForge, The New York Times [**Table 2.1**]. Cassandra is used in social websites like Digg, Facebook, Twitter [**Table 2.1**]. HBase is also used in Facebook [**Table 2.1**]. So, every database has usage in cloud computing. All we need to pick the appropriate database tool according to the need of application.

Chapter 4

Big Data Analytics:

Hadoop & MapReduce – A New Challenge

Big data is big news and so too analytics on big data. Technologies for analyzing big data are evolving rapidly and there is significant interest in new analytic approaches such as **Hadoop** and **MapReduce** [35]. We analyzed a newly evolving NOSQL database in previous chapter. Now, in this chapter, we aim to make investigation on Hadoop and MapReduce.

4.1 MapReduce

MapReduce is a technique popularized by Google that distributes the processing of very large multi-structured data files across a large cluster of machines [36]. High performance is achieved by breaking the processing into small units of work that can be run in parallel across the hundreds, potentially thousands, of nodes in the cluster.

To quote the seminal paper on MapReduce:

> "MapReduce is a programming model and an associated implementation for processing and generating large data sets. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system."

The key point to note from this quote is that MapReduce is a programming model, not a programming language, i.e., it is designed to be used by programmers, rather than business users.

So, if we have to then we can define MapReduce in one sentence as –

"MapReduce is a programming model for automating parallel computing."

4.1.1 Fundamental Pieces of MapReduce query

There are two fundamental pieces of a MapReduce query –

Map

The master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes [37]. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

Reduce

The master node then takes the answers to all the sub-problems and combines them in a way to get the output - the answer to the problem it was originally trying to solve.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows handling lists of values that are too large to fit in memory.

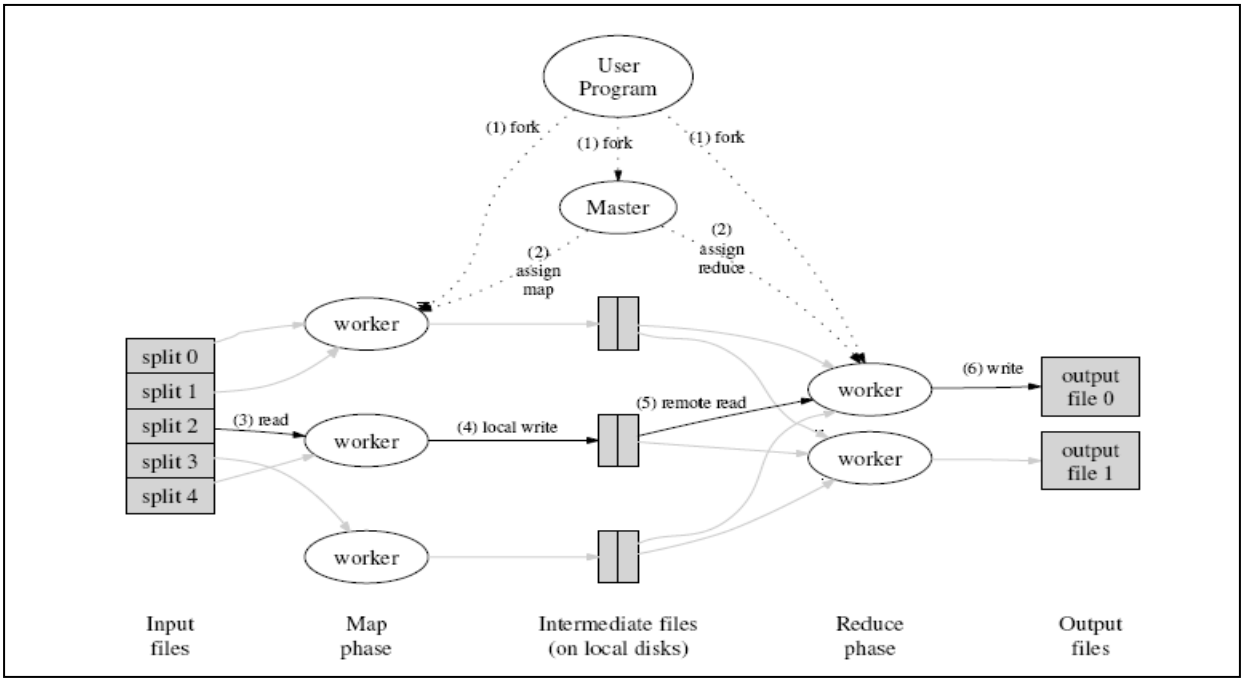


Figure 4.1 MapReduce Execution Overview

4.1.2 MapReduce Usage

MapReduce aids organizations in processing and analyzing large volumes of multi-structured data. Application examples include indexing and search, graph analysis, text analysis, machine learning, data transformation, and so forth. These types of applications are often difficult to implement using the standard SQL employed by relational DBMSs.

The procedural nature of MapReduce makes it easily understood by skilled programmers. It also has the advantage that developers do not have to be concerned with implementing parallel computing – this is handled transparently by the system. Although MapReduce is designed for programmers, non-programmers can exploit the value of prebuilt MapReduce applications and function libraries. Both commercial and open source MapReduce libraries are available that provide a wide range of analytic capabilities. Apache Mahout, for example, is an open source machine-learning library of "algorithms for clustering, classification and batch-based collaborative filtering" that are implemented using MapReduce.

4.1.3 Application Development

MapReduce programs are usually written in Java, but they can also be coded in languages such as C++, Perl, Python, Ruby, R, etc. These programs may process data stored in different file and database systems. At Google, for example, MapReduce was implemented on top of the Google File System (GFS).

One of the main deployment platforms for MapReduce is the open source Hadoop distributed computing framework provided by Apache Software Foundation. Hadoop supports MapReduce processing on several file systems, including the

Hadoop Distributed File System (HDFS), which was motivated by GFS. Hadoop also provides Hive and Pig, which are high-level languages that generate MapReduce programs. Several vendors offer open source and commercially supported Hadoop distributions; examples include Cloudera, DataStax, Hortonworks (a spinoff from Yahoo) and MapR. Many of these vendors have added their own extensions and modifications to the Hadoop open source platform.

Another direction of vendors is to support MapReduce processing in relational DBMSs. These are implemented as *in-database* analytic functions that can be used in SQL statements. These functions are run inside the database system, which enables them to benefit from the parallel processing capabilities of the DBMS. Supported in the Teradata Aster MapReduce Platform, the Aster Database provides a number of built-in MapReduce functions for use with SQL. It also includes an interactive development environment, Aster Developer Express, for programmers to create their own MapReduce functions.

4.2 Hadoop

As we have stated before, Google was the first to publicize MapReduce, a system they had used to scale their data processing needs. This system aroused a lot of interest because many other businesses were facing similar scaling challenges, and it wasn't feasible for everyone to reinvent their own proprietary tool. Doug Cutting[2] saw an opportunity and led the charge to develop an open source version of this MapReduce system called Hadoop, Yahoo and others rallied around to support this effort. Today, Hadoop is a core part of the computing infrastructure for many web

[2] Douglas Read Cutting is an advocate and creator of open-source search technology. He originated Lucene and, with Mike Cafarella, Nutch, both open-source search technology projects which are now managed through the Apache Software Foundation.

companies, such as Yahoo, Facebook, LinkedIn, and Twitter. Many more traditional businesses, such as media and telecom, are beginning to adopt this system too.
Here, we describe the fundamental idea of Hadoop.

Hadoop is a generic processing framework designed to execute queries and other batch read operations against massive datasets that can be tens or hundreds of terabytes and even petabytes in size. The data is loaded into or appended to the Hadoop Distributed File System (HDFS). Hadoop then performs brute force scans through the data to produce results that are output into other files. It probably does not qualify as a database since it does not perform updates or any transactional processing. Hadoop also does not support such basic functions as indexing or a SQL interface, although there are additional open source projects underway to add these capabilities.

Hadoop operates on massive datasets by horizontally scaling (aka scaling out) the processing across very large numbers of servers through an approach called MapReduce. Vertical scaling (aka scaling up), i.e., running on the most powerful single server available, is both very expensive and limiting. There is no single server available today or in the foreseeable future that has the necessary power to process so much data in a timely manner.

Figure 4.2 Clusters of machine running Hadoop at Yahoo! (Source: Yahoo!)

Hundreds or thousands of small, inexpensive, commodity servers do have the power if the processing can be horizontally scaled and executed in parallel. Using the MapReduce approach, Hadoop splits up a problem, sends the sub-problems to different servers, and lets each server solve its sub-problem in parallel. It then merges all the sub-problem solutions together and writes out the solution into files which may in turn be used as inputs into additional MapReduce steps.

Hadoop has been particularly useful in environments where massive server farms are being used to collect the data. Hadoop is able to process parallel queries as big, background batch jobs on the same server farm. This saves the user from having to acquire additional hardware for a database system to process the data. Most importantly, it also saves the user from having to load the data into another system. The huge amount of data that needs to be loaded can make this impractical.

4.2.1 What is Hadoop Good For

When the original MapReduce algorithms were released, and Hadoop was subsequently developed around them, these tools were designed for specific uses. The original use was for managing large data sets that needed to be easily searched. As time has progressed and as the Hadoop ecosystem has evolved, several other specific uses have emerged for Hadoop as a powerful solution.

In this part, we summarize Hadoop usage.

Large Data Sets

MapReduce paired with HDFS is a successful solution for storing large volumes of unstructured data.

Scalable Algorithms

Any algorithm that can scale too many cores with minimal inter-process communication will be able to exploit the distributed processing capability of Hadoop.

Log Management

Hadoop is commonly used for storage and analysis of large sets of logs from diverse locations. Because of the distributed nature and scalability of Hadoop, it creates a solid platform for managing, manipulating, and analyzing diverse logs from a variety of sources within an organization.

Extract-Transform-Load (ETL) Platform

Many companies today have a variety of data warehouse and diverse relational database management system (RDBMS) platforms in their IT environments. Keeping data up to date and synchronized between these separate platforms can be a struggle. Hadoop enables a single central location for data to be fed into, then processed by ETL-type jobs and used to update other, separate data warehouse environments.

4.2.2 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

HDFS is the file system component of Hadoop. HDFS stores file system metadata and application data separately. Hadoop has a variety of node types within each Hadoop cluster; these include DataNodes, NameNodes, and EdgeNodes. Names of these nodes can vary from site to site, but the functionality is common across the sites.

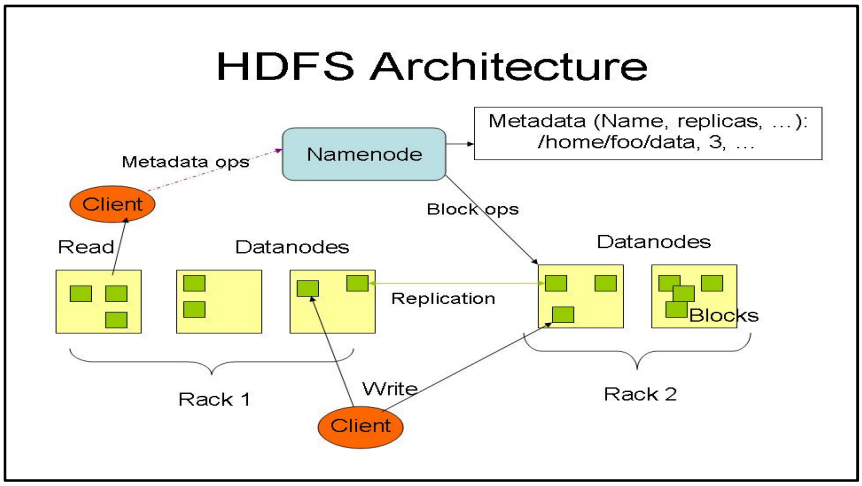


Figure 4.3 HDFS Architecture

The base node types for a Hadoop cluster are described below [38] –

NameNode

The NameNode is the central location for information about the file system deployed in a Hadoop environment. An environment can have one or two NameNodes, configured to provide minimal redundancy between the NameNodes. The NameNode is contacted by clients of the Hadoop Distributed File System (HDFS) to locate information within the file system and provide updates for data they have added, moved, manipulated, or deleted.

DataNode

DataNodes make up the majority of the servers contained in a Hadoop environment. Common Hadoop environments will have more than one DataNode, and oftentimes they will number in the hundreds based on capacity and performance needs. The DataNode serves two functions: It contains a portion of the data in the HDFS and it acts as a computing platform for running jobs, some of which will utilize the local data within the HDFS.

EdgeNode

The EdgeNode is the access point for the external applications, tools, and users that need to utilize the Hadoop environment. The EdgeNode sits between the Hadoop cluster and the corporate network to provide access control, policy enforcement, logging, and gateway services to the Hadoop environment. A typical Hadoop environment will have a minimum of one EdgeNode and more based on performance needs.

4.2.3 MapReduce in Hadoop

HDFS delivers inexpensive, reliable, and available file storage. That service alone, though, would not be enough to create the level of interest, or to drive the rate of adoption, that characterizes Hadoop over the past several years. The second major component of Hadoop is the parallel data processing system called MapReduce. Conceptually, MapReduce is simple.

MapReduce includes a software component called the job scheduler. The job scheduler is responsible for choosing the servers that will run each user job, and for scheduling execution of multiple user jobs on a shared cluster. The job scheduler consults the NameNode for the location of all of the blocks that make up the file or

files required by a job. Each of those servers is instructed to run the user's analysis code against its local block or blocks. The MapReduce processing infrastructure includes an abstraction called an *input split* that permits each block to be broken into individual records. There is special processing built in to reassemble records broken by block boundaries. The user code that implements a map job can be virtually anything. MapReduce allows developers to write and deploy code that runs directly on each DataNode server in the cluster. That code understands the format of the data stored in each block in the file, and can implement simple algorithms (count the number of occurrences of a single word, for example) or much more complex ones (e.g. natural language processing, pattern detection and machine learning, feature extraction, or face recognition).

At the end of the map phase of a job, results are collected and filtered by a *reducer*. MapReduce guarantees that data will be delivered to the reducer in sorted order, so output from all mappers is collected and passed through a *shuffle and sort* process. The sorted output is then passed to the reducer for processing. Results are typically written back to HDFS.

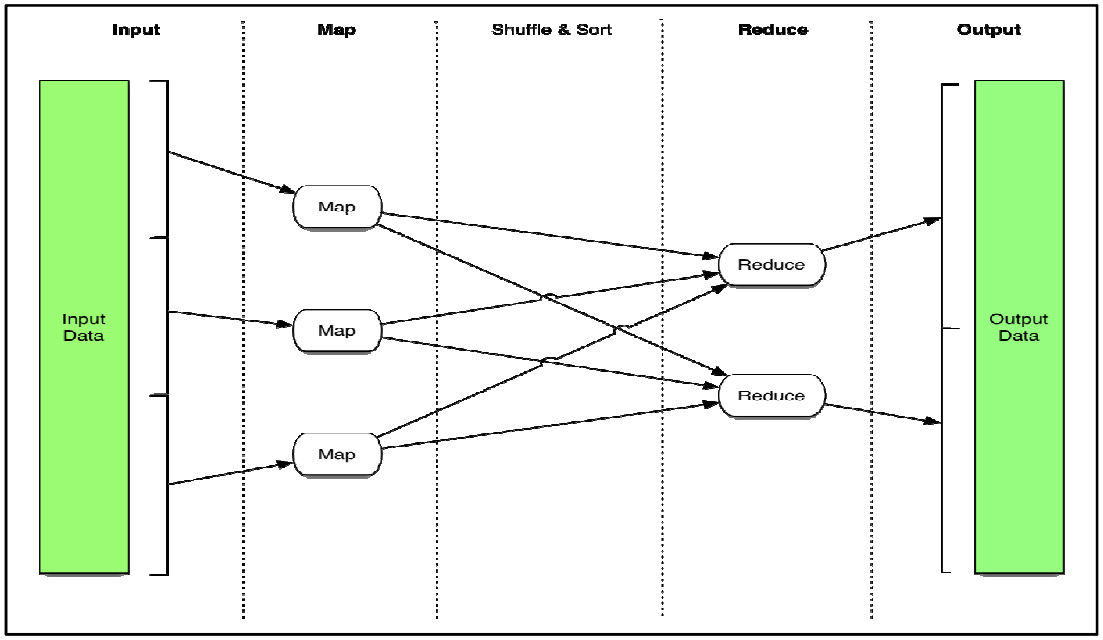


Figure 4.4 Model of Hadoop MapReduce

Because of the replication built into HDFS, MapReduce is able to provide some other useful features. For example, if one of the servers involved in a MapReduce job is running slowly — most of its peers have finished, but it is still working — the job scheduler can launch another instance of that particular task on one of the other servers in the cluster that stores the file block in question. This means that overloaded or failing nodes in a cluster need not stop, or even dramatically slow down, a MapReduce job.

An important part of our thesis includes Hadoop configuration and testing MapReduce function on Hadoop framework. We have performed this task on single node cluster. This paper shows the steps of 'Configuring Virtual Machines with Hadoop' in Appendix A and then, an experiment is included on 'Testing MapReduce Program' in Appendix B.

Chapter 5

Hive – Data Warehouse Using Hadoop

In the previous chapter, we investigated on popular Hadoop framework. One disadvantage of Hadoop is that it is not easy for end users, especially for the ones who are not familiar with map/reduce. Hadoop lacked the expressibility of popular query languages like SQL and as a result users ended up spending hours to write programs for typical analysis [39]. So, it is very clear that in order to really empower the companies to analyze their data more productively, it is necessary to improve the query capabilities of Hadoop. Bringing this data closer to users is what inspired to build **Hive**. This paper focuses on Hive because it has gained the most acceptance in the industry like Facebook and also because it's SQL-like syntax makes it easy to use by non-programmers who are comfortable using SQL.

In this chapter, we aim to investigate on Hive and make an experiment on Hive wrapper on top of Hadoop.

5.1 Hive

Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. The query language can be easily understood by anyone familiar with SQL. At the same time this language also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL.

5.2 Hive Architecture

The main components of Hive are illustrated in **Figure 5.1.** HiveQL statements can be entered using a command line or Web interface, or may be embedded in applications that use ODBC and JDBC interfaces to the Hive system. The Hive Driver system converts the query statements into a series of MapReduce jobs.

The main components of Hive are –

UI
The user interface for users to submit queries and other operations to the system. Currently the system has a command line interface and a web based GUI is being developed.

Driver

The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.

Compiler

The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the Metastore.

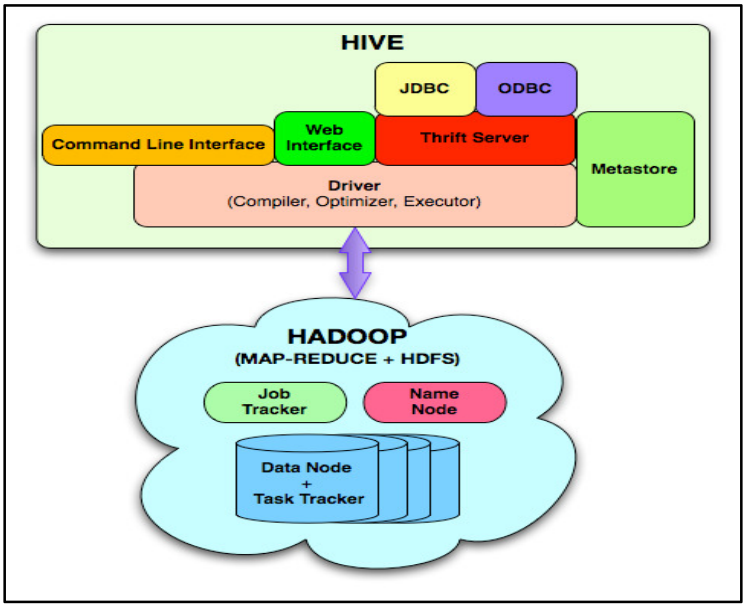


Figure 5.1 Hive Architecture

Metastore

The component that stores all the structure information of the various table and partitions in the warehouse including column and column type

information, the serializers and deserializers necessary to read and write data and the corresponding hdfs files where the data is stored.

Execution Engine

The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

5.3 Hive Data Models

Similar to traditional databases, Hive stores data in tables, where each table consists of a number of rows, and each row consists of a specified number of columns. We describe Hive data model components below –

Tables

These are analogous to Tables in Relational Databases. Tables can be filtered, projected, joined and unioned. Additionally all the data of a table is stored in a directory in hdfs. Hive also supports notion of external tables wherein a table can be created on preexisting files or directories in hdfs by providing the appropriate location to the table creation DDL. The rows in a table are organized into typed columns similar to Relational Databases. [40]

Partitions

Each Table can have one or more partition keys which determine how the data is stored e.g. a table T with a date partition column ds had files with data for a particular date stored in the <table location>/ds=<date> directory in hdfs.

Partitions allow the system to prune data to be inspected based on query predicates, e.g. a query that in interested in rows from T that satisfy the predicate T.ds = '2008-09-01' would only have to look at files in <table location>/ds=2008-09-01/ directory in hdfs.

Buckets

Data in each partition may in turn be divided into Buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory. Bucketing allows the system to efficiently evaluate queries that depend on a sample of data (these are queries that use SAMPLE clause on the table).

5.4 HiveQL in Hadoop

Data files in Hive are seen in the form of tables (and views) with columns to represent fields and rows to represent records. Tables can be vertically partitioned based on one or more table columns. The data for each partition is stored in a separate HDFS file. Data is not validated against the partition definition during the loading of data into the HDFS file. Partitions can be further split into buckets by hashing the data values of one or more partition columns. Buckets are stored in separate HDFS files and are used for sampling and for building a Hive index on an HDFS file. Hive tables do not support the concepts of primary or foreign keys or constraints of any type. All definitions are maintained in the Hive *metastore*, which is a relational database such as MySQL.

Data types supported by Hive include primitive types such as integers, floating point numbers, strings and Boolean. A timestamp data type is provided in Hive

0.8.0. Hive also supports complex types such as arrays (indexed lists), maps (key value pairs), structs (structures) and user-defined types.

External HDFS files can be defined to Hive as *external* tables. Hive also allows access to data stored in other file and database systems such as HBase. Access to these data stores is enabled via storage handlers that present the data to Hive as *non-native* tables. The HiveQL support (and restrictions) for these non-native tables is broadly the same as that for native tables.

HiveQL supports a subset of the SQL SELECT statement operators and syntax, including:

- Join (equality, outer, and left semi-joins are supported)
- Union
- Subqueries (supported in the FROM clause only)
- Relational, arithmetic, logical and complex type operators
- Arithmetic, string, date, XPath, and user-defined functions including aggregate and table functions (UDFs, UDAFs, UDTFs)

The Hive MAP and REDUCE operators can be used to embed custom MapReduce scripts in HiveQL queries. An INSERT statement is provided, but it can only be used to load or replace a complete table or table partition. The equivalents of the SQL UPDATE and DELETE statements are not supported.

When comparing Hadoop Hive to a relational DBMS employing SQL, two areas have to be considered: query language syntax and query performance. Query language syntax is a moving target in both Hadoop Hive and relational DBMS products. Although Hive provides a useful subset (and superset) of SQL

functionality, it is highly probable that existing SQL-based user applications and vendor software would need to be modified and simplified to work with Hive.

Perhaps the most important comparison between Hadoop Hive and the relational DBMS environment concerns performance. Such comparisons should consider traditional short and/or *ad-hoc* SQL-like queries running on Hive versus a relational DBMS, and also MapReduce performance on Hive compared with using SQL MapReduce relational DBMS functions for querying and analyzing large volumes of multi-structured data. Knowledge of the way Hive and relational DBMSs process queries is useful when discussing the performance of the two approaches.

Hive provides an SQL wrapper on top of Hadoop HDFS (and other storage systems). It has no control or knowledge of the placement or location of data in HDFS files. The Hive optimizer uses rules to convert HiveQL queries into a series of MapReduce jobs for execution on a Hadoop cluster. Hints in HiveQL queries can aid the optimization process, for example, to improve the performance of join processing.

Hive-generated MapReduce jobs sequentially scan the HDFS files used to store the data associated with a Hive table. The Hive optimizer is partition and bucket aware, and so table partitions and buckets can be used to reduce the amount of data scanned by a query. Hive supports compact indexes (in 0.7.0) and bitmapped indexes (in 0.8.0), which aid lookup and range queries, and also enable certain queries to be satisfied by index-only access. Since Hive has no knowledge of the actual physical location of the data in an HDFS file, the Hive indexes contain data rather than pointers to data. A table index will need to be rebuilt if the data in a table partition is refreshed. Hive does, however, support partitioned indexes. Hive indexes aid the performance of traditional SQL-like queries, rather than MapReduce queries, which by their nature involve sequential processing.

The primary use case for Hadoop Hive is the same as that for Hadoop MapReduce, which is the sequential processing of very large multi-structured data files such as Web logs. It is not well suited to *ad- hoc* queries where the user expects fast response times. The positioning of Hive is aptly described on the Apache Hive Wiki. "Hive is not designed for OLTP workloads and does not offer real-time queries or row-level updates. It is best used for batch jobs over large sets of append-only data (like Web logs). What Hive values most is scalability (scale out with more machines added dynamically to the Hadoop cluster), extensibility (with MapReduce framework and UDF/UDAF/UDTF), fault-tolerance, and loose-coupling with its input formats.

The main benefit of Hive is that it dramatically improves the simplicity of MapReduce development. The Hive optimizer also makes it easier to process interrelated files as compared with hand-coding MapReduce procedural logic to do this. The Hive optimizer, however, is still immature and is not fully insulated from the underlying file system, which means that for more complex queries, the Hive user is still frequently required to aid the optimizer through hints and certain HiveQL language constructions.

There are many other tools for improving the usability of Hadoop, e.g., Informatica HParser for data transformation, and Karmasphere Studio, Pentaho Business Analytics and Revolution RevoConnectR for analytical processing. Most of these tools are front ends to Hadoop MapReduce and HDFS, and so many of the considerations discussed above for Hive apply equally to these products.

This paper includes an experiment on 'Testing HiveQL on Hadoop framework'. Appendix C describes the steps of 'Configuration of Hive' and Appendix D presents the procedure of 'Testing HiveQL'.

Chapter 6

Discussion

Today, we're surrounded by data. People upload videos, take pictures on their cell phones, text friends, update their Facebook status, leave comments around the web, click on ads, and so forth. Machines, too, are generating and keeping more and more data. The exponential growth of data first presented challenges to cutting-edge businesses such as Google, Yahoo, Amazon, and Microsoft. They needed to go through exabytes and zettabytes of data to figure out which websites were popular, what books were in demand, and what kinds of ads appealed to people. Existing tools were becoming inadequate to process such large data sets. So, new technologies have emerged.

The idea behind this thesis is aimed at making investigation on newer approaches of cloud data storages. We have conducted the comparison study on 'SQL vs. NOSQL' issue that is considered as an important debate in cloud world. Then, we made research on different branches of newly evolving NOSQL database. NOSQL database contains a magnificent field in cloud computing. So, we picked 3 popular

NOSQL databases – MongoDB, Cassandra and HBase to explore. Later, a case study has been conducted to identify the distinct usage properties of each database.

In the next part of investigation, we also conducted an experiment on best knowing framework – Hadoop and data warehouse system – Hive. This combination of Hadoop - Hive was known to serve data storage of big website like Facebook.

There is no doubt that the work done here is the root of further research on NOSQL database and 'Big Data Analytics'. In future, we aim to make research on more NOSQL databases and perform a real-time benchmark test of NOSQL databases using YCSB in Cloud environment.

Also, in our thesis, we implemented Hadoop and Hive and evaluated MapReduce function and HiveQL. These works have been done on single-node-cluster. Our future plan from this part is to test HiveQL and MapReduce on multi-node-cluster.

Appendix A

Configuring Virtual Machines with Hadoop

In this appendix, the required steps for setting up a Hadoop single node cluster using the Hadoop Distributed File System (HDFS) on Ubuntu Linux will be described.

A.1 Running Hadoop on Single Node Cluster

This experiment has been tested with the following software version:

- Ubuntu Linux 11.04
- Hadoop 0.20.2
- Hive 0.8.1

Prerequisites

- At first we set up the Virtual Machine [Oracle VM Virtual Box]
- Then we install the Ubuntu version 11.04
- After that configure the NAT(it is important for getting net connection)

NAT Configuration

Machine-->Setting-->Network-->NAT-->OK (Done)

A.1.1. Changing root password

sumaiya@VMHadoop:~$ sudopasswd
[sudo] password for sumaiya:
Enter new UNIX password:
Retypenew UNIX password:
passwd: password updated successfully

```
[sudo] password for hadoopuser:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
hadoopuser@hadoopuser-VirtualBox:~$ su -sudo
Password:
```

A.1.2. Adding Hadoop User

#useraddhadoop
#mkdir /home/hadoop
#chown -R hadoop:hadoop /home/hadoop
#usermod -d /home/hadoophadoop

#passwdhadoop
newpassword:hadoop

A.1.3. Providing Sudo Privilege to Hadoop User

#usermod -a -G sudohadoop
#login as root
#chmodu+w /etc/sudoers
#vi /etc/sudoers

Inserting the below line into the File

```
hadoop        ALL=(ALL:ALL)ALL
```

A.1.4. Checking Hadoop's Sudo Privilege

```
#chmod u-w /etc/sudoers
#su - hadoop
//checking whether hadoop user is getting sudo privilege or not
$cat /etc/sudoers //this will show permission denied
$sud cat /etc/sudoers //this will work as hadoop user got sudo privilege
```

A.1.5. Changing shell for Hadoop User

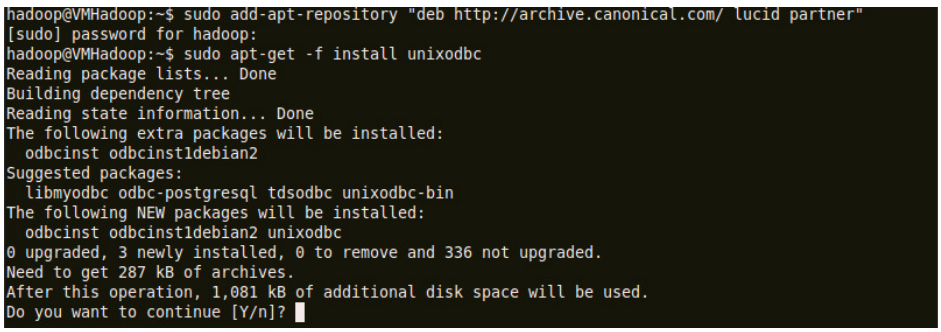
```
$sudo vi/etc/passwd

(change shell of hadoop from /bin/sh to /bin/bash)
```

A.1.6 Installing ODBC Components

```
$sudo add-apt-repository "deb http://archive.canonical.com/
lucid partner"

$ sudo apt-get -f install unixodbc
```



$ sudo apt-get -f install odbcinst1debian2

```
hadoop@VMHadoop:~$ sudo apt-get -f install odbcinst1debian2
Reading package lists... Done
Building dependency tree
Reading state information... Done
odbcinst1debian2 is already the newest version.
odbcinst1debian2 set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 336 not upgraded.
```

$ sudo apt-get -f install odbcinst

```
hadoop@VMHadoop:~$ sudo apt-get -f install odbcinst
Reading package lists... Done
Building dependency tree
Reading state information... Done
odbcinst is already the newest version.
odbcinst set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 336 not upgraded.
```

A.1.7. Installing Java

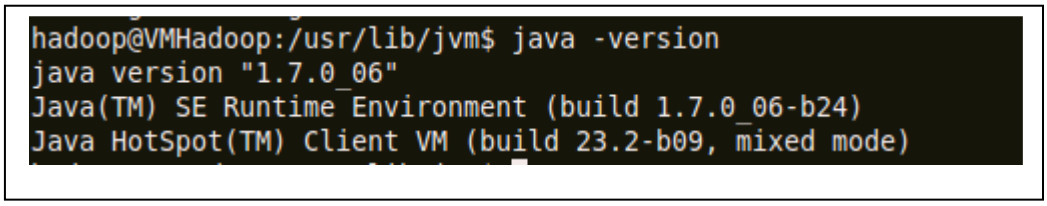
```
#mkdir /mnt/share                    //Creating a share mount point for host OS
#chown -R hadoop:hadoop /mnt/share
$sudo mount -t vboxsfhadoop_software /mnt/share
$ sudomkdir /usr/lib/jvm/                    //Creating JAVA Home Directory
$cd /mnt/share
$sudocp jdk-7u6-linux-i586.tar.gz /usr/lib/jvm      //copying java software to guest OS
from host OS

$ sudo tar zxvf jdk-7u6-linux-i586.tar.gz          //unzip the java software
$ sudo mv jdk-7u6-linux-i586 /usr/lib/jvm/

$ sudo update-alternatives --install /usr/bin/java java
/usr/lib/jvm/jdk1.7.0_06/bin/java 1
$ sudo update-alternatives --install /usr/bin/javacjavac
/usr/lib/jvm/jdk1.7.0_06/bin/javac 1

$ sudo update-alternatives --configjavac
$ sudo update-alternatives --config java
```

Quick Check for SUN JDK's Correct Set Up:



A.1.8. Configuring SSH

$ sudo add-apt-repository "deb http://archive.canonical.com/ lucid partner"

$ sudo apt-get upgrade openssh-client openssh-server

//Generate an SSH key for the hadoop:

$ su - hadoop
$ ssh-keygen -t rsa -P ""

```
hadoop@VMHadoop:/usr/local$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Created directory '/home/hadoop/.ssh'.
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
The key fingerprint is:
c9:57:8c:6e:d2:54:b5:22:68:6c:3d:b6:86:3e:18:06 hadoop@VMHadoop
The key's randomart image is:
+--[ RSA 2048]----+
|            ...  |
|        . o +   . |
|    E   = B + .  |
|     . + B = .   |
|      o S B      |
|     . + =       |
|      . o        |
|         .       |
|                 |
+-----------------+
hadoop@VMHadoop:/usr/local$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
hadoop@VMHadoop:/usr/local$
```

$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys

```
hadoop@VMHadoop:/usr/local$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
hadoop@VMHadoop:/usr/local$ sudo vi /etc/sysctl.conf
[sudo] password for hadoop:
hadoop@VMHadoop:/usr/local$
```

$ sudo apt-get install openssh-server openssh-client

```
hadoop@VMHadoop:~$ sudo apt-get install openssh-server openssh-client
Reading package lists... Done
Building dependency tree
Reading state information... Done
openssh-client is already the newest version.
Suggested packages:
  rssh molly-guard openssh-blacklist openssh-blacklist-extra
The following NEW packages will be installed:
  openssh-server ssh-import-id
0 upgraded, 2 newly installed, 0 to remove and 7 not upgraded.
Need to get 317 kB of archives.
After this operation, 913 kB of additional disk space will be used.
WARNING: The following packages cannot be authenticated!
  openssh-server ssh-import-id
Install these packages without verification [y/N]? y
Get:1 http://bd.archive.ubuntu.com/ubuntu/ natty/main openssh-server i386 1:5.8p1-1ubuntu3 [311 kB]
Get:2 http://bd.archive.ubuntu.com/ubuntu/ natty/main ssh-import-id all 2.4-0ubuntu1 [5,934 B]
Fetched 317 kB in 13s (22.7 kB/s)
Preconfiguring packages ...
Selecting previously deselected package openssh-server.
(Reading database ... 132075 files and directories currently installed.)
Unpacking openssh-server (from .../openssh-server_1%3a5.8p1-1ubuntu3_i386.deb) ...
Selecting previously deselected package ssh-import-id.
Unpacking ssh-import-id (from .../ssh-import-id_2.4-0ubuntu1_all.deb) ...
Processing triggers for ureadahead ...
ureadahead will be reprofiled on next reboot
Processing triggers for ufw ...
Processing triggers for man-db ...
Setting up openssh-server (1:5.8p1-1ubuntu3) ...
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Creating SSH2 ECDSA key; this may take some time ...
```

//now below will not prompt for password
$ sshlocalhost

```
hadoop@VMHadoop:~$ ssh localhost
Welcome to Ubuntu 11.04 (GNU/Linux 2.6.38-8-generic i686)

 * Documentation:  https://help.ubuntu.com/

Last login: Wed Dec  5 02:55:35 2012 from localhost
```

A.1.9. Disabling IPV6

//Open /etc/sysctl.conf and add the following lines to the end of the file:

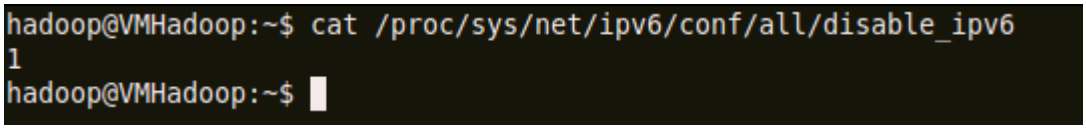
$sudovi /etc/sysctl.conf

#disable ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1

//Reboot the machine in order to make the changes take effect.

//You can check whether IPv6 is enabled on your machine with the following command:

$ cat /proc/sys/net/ipv6/conf/all/disable_ipv6
1



A.1.10. Installing and Configuring Hadoop:

$sudo mount -t vboxsfhadoop_software /mnt/share
//extract and change ownership to hadoop user
$cd /mnt/share
$sudocphadoop-0.20.2.tar.gz/usr/local
$cd /usr/local
$sudochownhadoop:hadoophadoop-0.20.2.tar.gz
$sudo tar xzf hadoop-0.20.2.tar.gz
$sudo mv hadoop-0.20.2 hadoop
$sudochown -R hadoop:hadoophadoop

```
root@VMHadoop:/usr/local# chown -R hadoop:hadoop hadoop
root@VMHadoop:/usr/local# ls -ltrh
total 40K
drwxr-xr-x 12 hadoop hadoop 4.0K 2010-02-19 14:10 hadoop
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 src
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 sbin
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 include
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 games
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 etc
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 bin
drwxr-xr-x  3 root   root   4.0K 2011-04-26 04:51 lib
lrwxrwxrwx  1 root   root      9 2012-12-04 13:47 man -> share/
drwxr-xr-x  8 root   root   4.0K 2012-12-05 00:39 share
drwxr-xr-x  9 root   root   4.0K 2012-12-05 01:15 hive
hadoop@VMHadoop:/mnt/share$ sudo cp hadoop-0.20.2.tar.gz /usr/local/
hadoop@VMHadoop:/mnt/share$ sudo cp hive-0.8.1.tar.gz /usr/local/
hadoop@VMHadoop:/mnt/share$ cd /usr/local
hadoop@VMHadoop:/usr/local$ ls -ltrh
total 73M
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 src
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 sbin
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 include
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 games
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 etc
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 bin
drwxr-xr-x 3 root root 4.0K 2011-04-26 04:51 lib
lrwxrwxrwx 1 root root    9 2012-12-04 13:47 man -> share/man
drwxr-xr-x 8 root root 4.0K 2012-12-05 00:39 share
-rwxr-xr-x 1 root root  43M 2012-12-05 01:02 hadoop-0.20.2.tar.gz
-rwxr-xr-x 1 root root  30M 2012-12-05 01:02 hive-0.8.1.tar.gz
hadoop@VMHadoop:/usr/local$ sudo tar xzf hadoop-0.20.2.tar.gz
hadoop@VMHadoop:/usr/local$ ls -ltrh
```

A.1.11. Setting Profile for Hadoop User

//Add the following lines to the end of the $HOME/.bashrc file of user hadoopuser:

```
##########hadoop related config#############
# Set Hadoop-related environment variables
export HADOOP_HOME=/usr/local/hadoop
export HIVE_HOME=/usr/local/hive

# Set JAVA_HOME (we will also configure JAVA_HOME directly for
Hadoop later on)
#export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_05/
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_06/

# Some convenient aliases and functions for running Hadoop-related
commands
unaliasfs&> /dev/null
aliasfs="hadoopfs"
unaliashls&> /dev/null
aliashls="fs -ls"

# If you have LZO compression enabled in your Hadoop cluster and
# compress job outputs with LZOP (not covered in this tutorial):
# Conveniently inspect an LZOP compressed file from the command
# line; run via:
#
# $ lzohead /hdfs/path/to/lzop/compressed/file.lzo
#
# Requires installed 'lzop' command.
#
lzohead () {
hadoopfs -cat $1 | lzop -dc | head -1000 | less
}

# AddHadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin
###############################################
```

```
hadoop@VMHadoop:~$ vi .bashrc
hadoop@VMHadoop:~$ . .bashrc
hadoop@VMHadoop:~$ echo $HIVE_HOME
/usr/local/hive
hadoop@VMHadoop:~$
```

A.1.12. Editing Hadoop's hadoop-eng.sh file

$sudovi /usr/local/hadoop/conf/hadoop-env.sh

//Change:
The java implementation to use. Required.
export JAVA_HOME=/usr/lib/j2sdk1.5-sun

//To:
The java implementation to use. Required.
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_06/

```
hadoop@VMHadoop:~$ sudo vi /usr/local/hadoop/conf/hadoop-env.sh
[sudo] password for hadoop:
hadoop@VMHadoop:~$ sudo mkdir -p /app/hadoop/tmp
hadoop@VMHadoop:~$ sudo chown hadoop:hadoop /app/hadoop/tmp
hadoop@VMHadoop:~$
```

A.1.13. Editing necessary XML files

//Add the following snippets between the <configuration> ... </configuration> tags in the respective configuration XML file.

$sudovi /usr/local/hadoop/conf/core-site.xml

<!-- In: conf/core-site.xml -->
<property>
<name>hadoop.tmp.dir</name>
<value>/app/hadoop/tmp</value>
<description>A base for other temporary directories.</description>
</property>

<property>
<name>fs.default.name</name>
<value>hdfs://localhost:54310</value>
<description>The name of the default file system. A URI whose scheme and authority determine the FileSystem implementation. The uri's scheme determines the config property (fs.SCHEME.impl) naming theFileSystem implementation class. The uri's authority is used to determine the host, port, etc. for a filesystem.</description>
</property>

//Add the following snippets between the <configuration> ... </configuration> tags in the respective configuration XML file.

$sudovi /usr/local/hadoop/conf/mapred-site.xml

```
<!-- In: conf/mapred-site.xml -->
<property>
<name>mapred.job.tracker</name>
<value>localhost:54311</value>
<description>The host and port that the MapReduce job tracker runs
at.  If "local", then jobs are run in-process as a single map
and reduce task.
</description>
</property>
```

//Add the following snippets between the <configuration> ... </configuration> tags in the respective configuration XML file.

$sudovi /usr/local/hadoop/conf/hdfs-site.xml

```
<!-- In: conf/hdfs-site.xml -->
<property>
<name>dfs.replication</name>
<value>1</value>
<description>Default block replication.
  The actual number of replications can be specified when the file is created.
  The default is used if replication is not specified in create time.
</description>
</property>
```

A.1.13. Creating Directory

//Now we create the directory and set the required ownerships and permissions:

$ sudomkdir -p /app/hadoop/tmp
$ sudochownhadoop:hadoop /app/hadoop/tmp

```
hadoop@VMHadoop:~$ sudo vi /usr/local/hadoop/conf/hadoop-env.sh
[sudo] password for hadoop:
hadoop@VMHadoop:~$ sudo mkdir -p /app/hadoop/tmp
hadoop@VMHadoop:~$ sudo chown hadoop:hadoop /app/hadoop/tmp
hadoop@VMHadoop:~$
```

A.1.14. Formatting Name Node

//Formatting the name node.

//Do not format a running Hadoopfilesystem as you will lose all the data currently in the cluster (in //HDFS).To format the filesystem (which simply initializes the directory specified by the dfs.name.dir//variable), run the command.

$/usr/local/hadoop/bin/hadoopnamenode –format

```
hadoop@VMHadoop:/usr/local/hadoop/bin$ hadoop namenode -format
12/12/05 02:39:03 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = VMHadoop/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.20.2
STARTUP_MSG:   build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20 -r 911707; compiled by 'chrisdo' on Fri Feb 19 08:07:34 UTC 2010
************************************************************/
12/12/05 02:39:04 INFO namenode.FSNamesystem: fsOwner=hadoop,hadoop,sudo
12/12/05 02:39:04 INFO namenode.FSNamesystem: supergroup=supergroup
12/12/05 02:39:04 INFO namenode.FSNamesystem: isPermissionEnabled=true
12/12/05 02:39:05 INFO common.Storage: Image file of size 96 saved in 0 seconds.
12/12/05 02:39:05 INFO common.Storage: Storage directory /app/hadoop/tmp/dfs/name has been successfully formatted.
12/12/05 02:39:05 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at VMHadoop/127.0.1.1
************************************************************/
hadoop@VMHadoop:/usr/local/hadoop/bin$
```

A.1.15 Starting Single-node Cluster

$ /usr/local/hadoop/bin ./start-all.sh

```
hadoop@VMHadoop:/usr/local/hadoop/bin$ ./start-all.sh
starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-namenode-VMHadoop.out
localhost: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-datanode-VMHadoop.out
localhost: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-secondarynamenode-VMHadoop.out
starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-jobtracker-VMHadoop.out
localhost: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-tasktracker-VMHadoop.out
hadoop@VMHadoop:/usr/local/hadoop/bin$
```

A.1.16 A notify tool for checking whether the expected Hadoop processes are running is JPS

$ cd /usr/lib/jvm/jdk1.7.0_06/bin

hadoop@VMHadoop:/usr/lib/jvm/jdk1.7.0_06/bin$ ./jps

```
hadoop@VMHadoop:/usr/lib/jvm/jdk1.7.0_06/bin$ ./jps
2287 DataNode
2676 TaskTracker
2885 Jps
2451 SecondaryNameNode
2137 NameNode
2514 JobTracker
hadoop@VMHadoop:/usr/lib/jvm/jdk1.7.0_06/bin$
```

```
hadoop@VMHadoop:/app/hadoop/tmp$ sudo netstat -plten | grep java
[sudo] password for hadoop:
tcp        0      0 0.0.0.0:34143           0.0.0.0:*               LISTEN      1001       18611      2514/java
tcp        0      0 0.0.0.0:50020           0.0.0.0:*               LISTEN      1001       19705      2287/java
tcp        0      0 0.0.0.0:41125           0.0.0.0:*               LISTEN      1001       18339      2451/java
tcp        0      0 127.0.0.1:54310         0.0.0.0:*               LISTEN      1001       17534      2137/java
tcp        0      0 127.0.0.1:54311         0.0.0.0:*               LISTEN      1001       18907      2514/java
tcp        0      0 0.0.0.0:50090           0.0.0.0:*               LISTEN      1001       19632      2451/java
tcp        0      0 0.0.0.0:50060           0.0.0.0:*               LISTEN      1001       19629      2676/java
tcp        0      0 127.0.0.1:38125         0.0.0.0:*               LISTEN      1001       19676      2676/java
tcp        0      0 0.0.0.0:50030           0.0.0.0:*               LISTEN      1001       19541      2514/java
tcp        0      0 0.0.0.0:50070           0.0.0.0:*               LISTEN      1001       19227      2137/java
tcp        0      0 0.0.0.0:59990           0.0.0.0:*               LISTEN      1001       16994      2137/java
tcp        0      0 0.0.0.0:50010           0.0.0.0:*               LISTEN      1001       19415      2287/java
tcp        0      0 0.0.0.0:39578           0.0.0.0:*               LISTEN      1001       17471      2287/java
tcp        0      0 0.0.0.0:50075           0.0.0.0:*               LISTEN      1001       19633      2287/java
hadoop@VMHadoop:/app/hadoop/tmp$
```

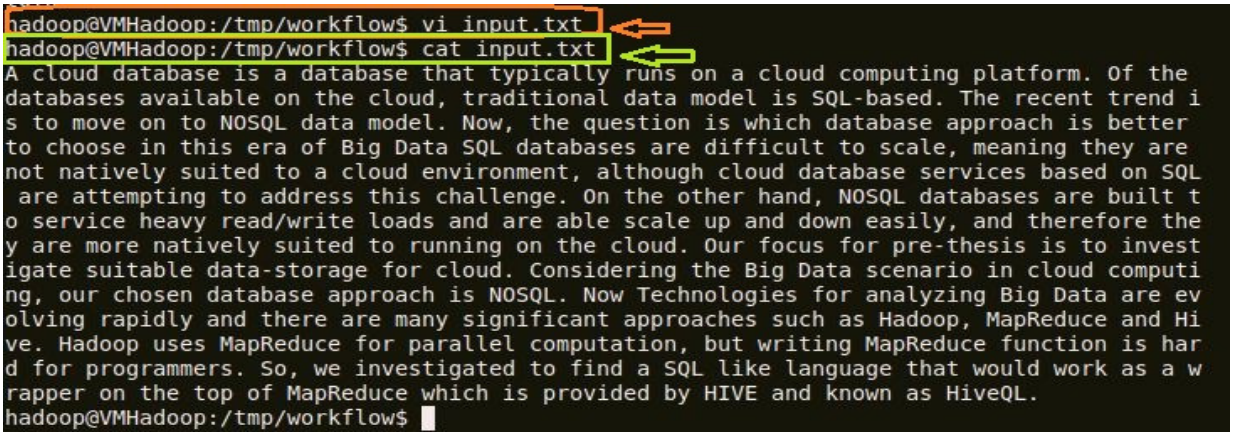
Appendix B

Testing MapReduce Program

In this Appendix we will run a "wordcount" problem to test the MapReduce function of Hadoop. This function will return number of occurrences of each word from a given text input file.

B.1. Running a MapReduce job

$ sudo mkdir /tmp/newout
$ cd /tmp
$ vi/tmp/newout/input.txt



$ /usr/local/hadoop/bin/hadoopdfs -copyFromLocal /tmp/newout/ /user/hduser/hadoopout

$ bin/hadoop jar hadoop *examples*jar wordcount/user/hduser/hadoop/input.txt/user/hduser/hadoop-count-out/

```
hadoop@VMHadoop:/usr/local/hadoop$ bin/hadoop jar hadoop*examples*.jar wordcount /user/hduser/hadoopout/input.txt /user/hduser/hadoop
12/12/11 00:35:46 INFO input.FileInputFormat: Total input paths to process : 1
12/12/11 00:35:47 INFO mapred.JobClient: Running job: job_201212102348_0003
12/12/11 00:35:48 INFO mapred.JobClient:  map 0% reduce 0%
12/12/11 00:36:07 INFO mapred.JobClient:  map 100% reduce 0%
12/12/11 00:36:24 INFO mapred.JobClient:  map 100% reduce 100%
12/12/11 00:36:30 INFO mapred.JobClient: Job complete: job_201212102348_0003
12/12/11 00:36:30 INFO mapred.JobClient: Counters: 17
12/12/11 00:36:30 INFO mapred.JobClient:   Job Counters
12/12/11 00:36:30 INFO mapred.JobClient:     Launched reduce tasks=1
12/12/11 00:36:30 INFO mapred.JobClient:     Launched map tasks=1
12/12/11 00:36:30 INFO mapred.JobClient:     Data-local map tasks=1
12/12/11 00:36:30 INFO mapred.JobClient:   FileSystemCounters
12/12/11 00:36:30 INFO mapred.JobClient:     FILE_BYTES_READ=1611
12/12/11 00:36:30 INFO mapred.JobClient:     HDFS_BYTES_READ=1236
12/12/11 00:36:30 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=3254
12/12/11 00:36:30 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=1110
12/12/11 00:36:30 INFO mapred.JobClient:   Map-Reduce Framework
12/12/11 00:36:30 INFO mapred.JobClient:     Reduce input groups=124
12/12/11 00:36:30 INFO mapred.JobClient:     Combine output records=124
12/12/11 00:36:30 INFO mapred.JobClient:     Map input records=1
12/12/11 00:36:30 INFO mapred.JobClient:     Reduce shuffle bytes=0
12/12/11 00:36:30 INFO mapred.JobClient:     Reduce output records=124
12/12/11 00:36:30 INFO mapred.JobClient:     Spilled Records=248
12/12/11 00:36:30 INFO mapred.JobClient:     Map output bytes=2055
12/12/11 00:36:30 INFO mapred.JobClient:     Combine input records=205
12/12/11 00:36:30 INFO mapred.JobClient:     Map output records=205
12/12/11 00:36:30 INFO mapred.JobClient:     Reduce input records=124
hadoop@VMHadoop:/usr/local/hadoop$
```

B.1.2. Retrieve the job result from HDFS

$ cd /usr/local/hadoop
$ mkdir /tmp/hdout
$ bin/hadoopdfs -getmerge /user/hduser/hadoop-count-out /tmp/hdout
$ head /tmp/hdout/hadoop-count-out

```
hadoop@VMHadoop:/usr/local/hadoop$ /usr/local/hadoop/bin/hadoop dfs -cat /user/hduser/hadoop-count-out/part-r-00000
A       1
Big     3
Considering     1
Data    3
HIVE    1
Hadoop  1
Hadoop, 1
Hive.   1
HiveQL. 1
MapReduce       4
NOSQL   2
NOSQL.  1
Now     1
Now,    1
Of      1
On      1
Our     1
```

Appendix C

Configuration of Hive

In this appendix, we will configure the Data WareHouse “Hive” on top of Hadoop.

Prerequisite

- Installation of Hadoop[We have used version 0.20.2]
- Need to download Hive [hive-0.8.1.tar.gz] from http://mirrors.ispros.com.bd/apache/hive/stable/

C.1. Installation of Hive

. $sudo cp hive-0.8.1.tar.gz /usr/local/

```
hadoop@VMHadoop:/mnt/share$ sudo cp hadoop-0.20.2.tar.gz /usr/local/
hadoop@VMHadoop:/mnt/share$ sudo cp hive-0.8.1.tar.gz /usr/local/
hadoop@VMHadoop:/mnt/share$ cd /usr/local
hadoop@VMHadoop:/usr/local$ ls -ltrh
total 73M
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 src
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 sbin
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 include
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 games
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 etc
drwxr-xr-x 2 root root 4.0K 2011-04-26 04:50 bin
drwxr-xr-x 3 root root 4.0K 2011-04-26 04:51 lib
lrwxrwxrwx 1 root root    9 2012-12-04 13:47 man -> share/man
drwxr-xr-x 8 root root 4.0K 2012-12-05 00:39 share
-rwxr-xr-x 1 root root  43M 2012-12-05 01:02 hadoop-0.20.2.tar.gz
-rwxr-xr-x 1 root root  30M 2012-12-05 01:02 hive-0.8.1.tar.gz
hadoop@VMHadoop:/usr/local$ sudo tar xzf hadoop-0.20.2.tar.gz
hadoop@VMHadoop:/usr/local$ ls -ltrh
```

$sudotar -xzvfhive-0.8.1.tar.gz

```
hadoop@VMHadoop:/usr/local$ sudo tar xzf hive-0.8.1.tar.gz
hadoop@VMHadoop:/usr/local$ ls -ltrh
total 30M
drwxr-xr-x 12 root root 4.0K 2010-02-19 14:10 hadoop-0.20.2
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 src
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 sbin
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 include
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 games
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 etc
drwxr-xr-x  2 root root 4.0K 2011-04-26 04:50 bin
drwxr-xr-x  3 root root 4.0K 2011-04-26 04:51 lib
lrwxrwxrwx  1 root root    9 2012-12-04 13:47 man -> share/man
drwxr-xr-x  8 root root 4.0K 2012-12-05 00:39 share
-rwxr-xr-x  1 root root  30M 2012-12-05 01:02 hive-0.8.1.tar.gz
drwxr-xr-x  9 root root 4.0K 2012-12-05 01:15 hive-0.8.1
hadoop@VMHadoop:/usr/local$ sudo rm hive-0.8.1.tar.gz
```

mv /usr/local/hive-0.8.1.tar.gz /usr/local/hive

chown -R hadoop:hadoop /usr/local/hive

```
root@VMHadoop:/usr/local# chown -R hadoop:hadoop hive
root@VMHadoop:/usr/local# ls -ltrh
total 40K
drwxr-xr-x 12 hadoop hadoop 4.0K 2010-02-19 14:10 hadoop
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 src
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 sbin
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 include
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 games
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 etc
drwxr-xr-x  2 root   root   4.0K 2011-04-26 04:50 bin
drwxr-xr-x  3 root   root   4.0K 2011-04-26 04:51 lib
lrwxrwxrwx  1 root   root      9 2012-12-04 13:47 man -> share/
drwxr-xr-x  8 root   root   4.0K 2012-12-05 00:39 share
drwxr-xr-x  9 hadoop hadoop 4.0K 2012-12-05 01:15 hive
```

C.2. Adding the below environment variables in ~/.bashrc file

```
$ export HIVE_HOME=/usr/local/hive

 $ export PATH=$HIVE_HOME/bin:$PATH
```

C.3. Configure hadoop HDFS before a table can be created in Hive

$HADOOP_HOME/bin/hadoopfs -mkdir /hivetest

```
hadoop@VMHadoop:/usr/local/hadoop/bin$ ./hadoop fs -mkdir  /hivetest
```

$HADOOP_HOME/bin/hadoopfs -chmodg+w /hivetest

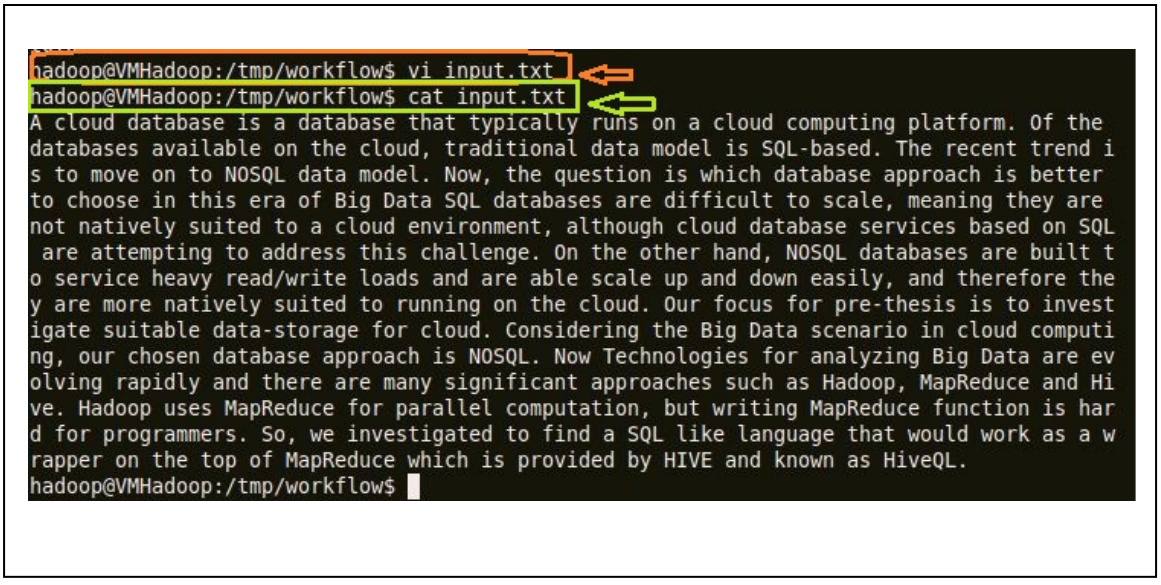
```
hadoop@VMHadoop:/usr/local/hadoop/bin$ ./hadoop fs -chmod g+w   /hivetest
hadoop@VMHadoop:/usr/local/hadoop/bin$
```

Appendix D

Testing HiveQL

In this appendix, we will test the data warehouse system Hive by running HiveQl on it. Here it will take input from a txt file “foo.txt” then it will load the data in table “foo”. There we can run our SQL like language which is HiveQL.

D.1. Sample text input file



D.2. Create table “hive_test”:

create table hive_test (word STRING,count INT) ROW FORMAT delimited fields terminated by '\t' lines terminated by '\n';

```
hive> create table hive_test (word STRING,count INT) ROW FORMAT delimited fields terminated by '\t' lines terminated by '\n';
OK
Time taken: 1.376 seconds
```

D.3. Loading output generated by MapReduce to HIVE’s “hive_test” Table:

hive> LOAD DATA LOCAL INPATH '/tmp/hdout/hadoop-out' OVERWRITE INTO TABLE hive_test;

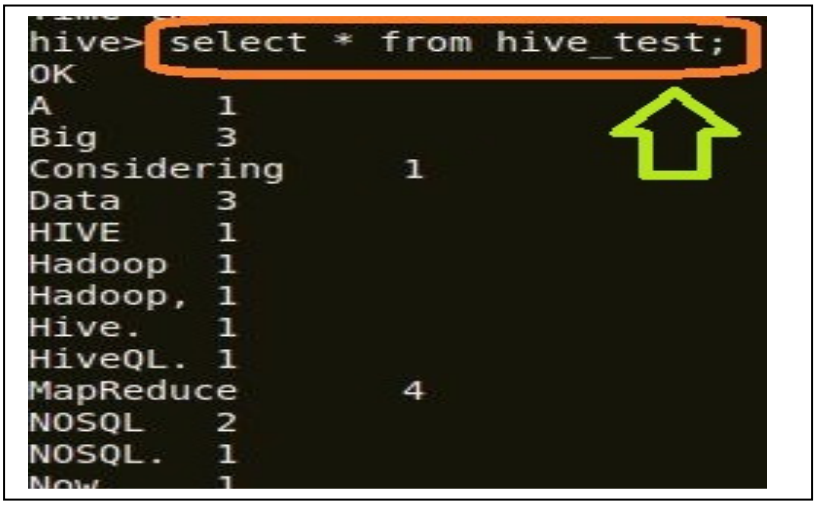
```
hive> LOAD DATA LOCAL INPATH '/tmp/hdout/hadoop-count-out' OVERWRITE INTO TABLE hive_test;
Copying data from file:/tmp/hdout/hadoop-count-out
Copying file: file:/tmp/hdout/hadoop-count-out
Loading data to table default.hive_test
Deleted hdfs://localhost:54310/user/hive/warehouse/hive_test
OK
Time taken: 1.862 seconds
```

D.4.1. HiveQL sample query example:

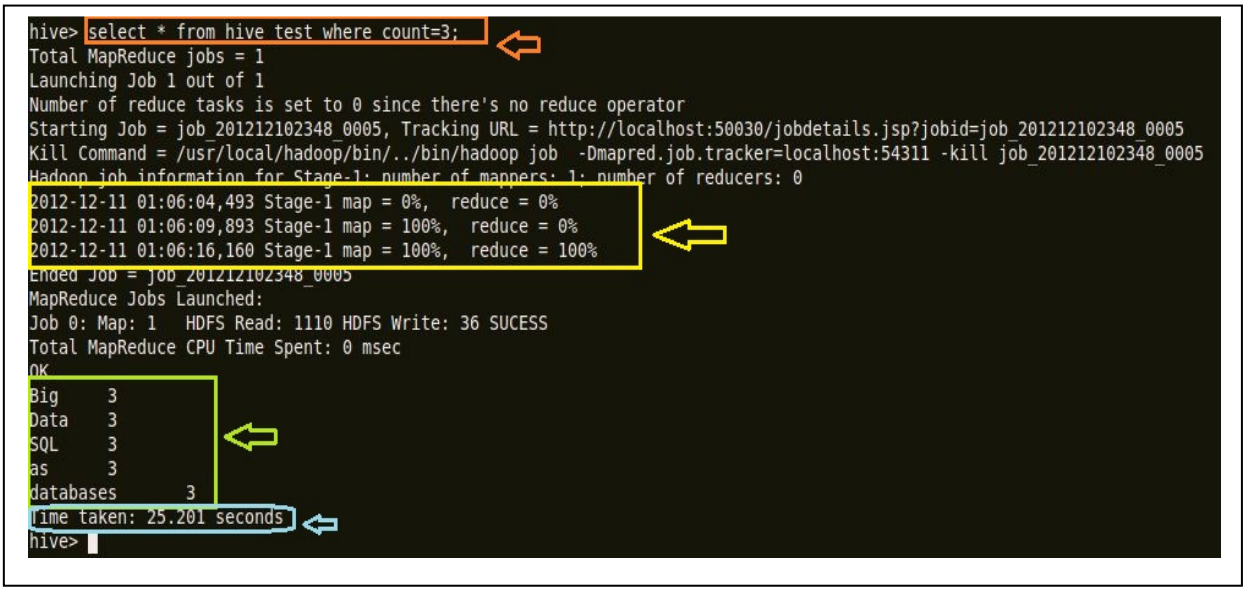
// Show Table:

```
hive> show tables;
OK
foo
hive_test
Time taken: 1.125 seconds
hive>
```

// select * from hive_test;



// Data Filtering: select * from hive_test where count=3;



List of Figures

List of Tables

Bibliography

[1] *NOSQL*.
http://www.slideshare.net/theburningmonk/introduction-to-nosql-_12025925

[2] *NoSQL. In the Context of Social Web*.
http://www.slideshare.net/hurrycane/nosql-in-the-context-of-social-web-4348152

[3] Wikibon (August 1, 2012). *A Comprehensive List of Big Data Statistics.*
http://wikibon.org/blog/big-data-statistics/

[4] *NoSQL.* In *Wikipedia*. Retrieved August 7, 2012, from
http://en.wikipedia.org/wiki/NoSQL

[5] Graph Database News Editor (July 7, 2012). *Not Only SQL, Not Only Hadoop.*
http://www.neotechnology.com/2012/06/not-only-sql-not-only-hadoop/

[6] A Monash Research Publication (July 31, 2011). *Terminology: Dynamic-vs. fixed-schema databases.*
http://www.dbms2.com/2011/07/31/dynamic-fixed-schema-databases/

[7] *NoSQL doesn't mean non-relational.*
http://www.xaprb.com/blog/2010/03/08/nosql-doesnt-mean-non-relational/

[8] *More on Non-relational (aka, NoSQL) databases.* (July 29, 2011).
http://www.larryullman.com/2011/07/29/more-on-non-relational-aka-nosql-databases/

[9] M. Loukides (February 8, 2012). *The NoSQL Movement.* http://strata.oreilly.com/2012/02/nosql-non-relational-database.html

[10] John D. Cook (July 6, 2009). *ACID versus BASE for database transactions.* http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/

[11] C. Roe (March 1, 2012). *ACID vs. BASE: The Shifting pH of Database Transaction Processing.* http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/

[12] *CAP theorem*. In *Wikipedia*. Retrieved August 8, 2012, from http://en.wikipedia.org/wiki/CAP_theorem

[13] C. Wilson (December 19, 2011). *Brewer's CAP Theorem.* http://craiggwilson.wordpress.com/2011/12/19/brewers-cap-theorem/

[14] *Scalability.* http://www.investopedia.com/terms/s/scalability.asp#axzz1wbooYu99

[15] M. Kopp (October 5, 2011). *NoSQL or RDBMS? – Are we asking the right questions?* http://blog.dynatrace.com/2011/10/05/nosql-or-rdbms-are-we-asking-the-right-questions/

[16] F. Firdausillah (March 09, 2011). *Database Availability and Integrity in NoSQL.* http://www.slideshare.net/kaqfa/nosql-availability-integrity

[17] *Cassandra.* http://lineofthought.com/tools/cassandra

[18] *MongoDB.* http://lineofthought.com/tools/mongodb

[19] *Apache HBase.* http://lineofthought.com/tools/apache-hbase

[20] *Redis.* http://lineofthought.com/tools/redis

[21] *The four categories of NoSQL databases.* http://rebelic.nl/engineering/the-four-categories-of-nosql-databases/

[22] M. Vardanyan (May 22, 2011). *Picking the Right NoSQL Tool.* http://blog.monitis.com/index.php/2011/05/22/picking-the-right-nosql-database-tool/

[23] R. Rees. *NoSQL, no problem.* http://www.thoughtworks.com/articles/nosql-comparison

[24] H. Kauhanen (March 09, 2010). *NoSQL database.* http://www.slideshare.net/harrikauhanen/nosql-3376398

[25] Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; *Benchmarking Cloud Serving Systems with YCSB.* In *ACM Symposium on Cloud Computing, ACM, Indianapolis, IN, USA (2010)*. http://research.yahoo.com/node/3202

[26] https://github.com/brianfrankcooper/YCSB/wiki

[27] https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload

[28] H. J. Lee. *NoSQL Benchmarking.* http://www.cubrid.org/blog/dev-platform/nosql-benchmarking/

[29] Ricky Ho (October 06, 2010). *Pragmatic Programming Techniques.* http://horicky.blogspot.com/2010/10/bigtable-model-with-cassandra-and-hbase.html

[30] *Cassandra Write Operation Performance Explained.* http://nosql.mypopescu.com/post/454521259/cassandra-write-operation-performance-explained

[31] M. Perham (March 17, 2010). *Cassandra Internals – Writing.* http://nosql.mypopescu.com/post/454521259/cassandra-write-operation performance-explained

[32] *Cassandra Reads Performance Explained.* http://nosql.mypopescu.com/post/474623402/cassandra-reads-performance-explained

[33] M. Perham (March 17, 2010). *Cassandra Internals – Reading.* http://www.mikeperham.com/2010/03/17/cassandra-internals-reading/

[34] A. Baranau (July 16, 2012). *Configuring HBase Memstore: What You Should Know.* http://blog.sematext.com/tag/architecture/

[35] J. Kelly. *Big Data: Hadoop, Business Analytics and Beyond.* http://wikibon.org/wiki/v/Big_Data:_Hadoop,_Business_Analytics_and_Beyond

[36] C. White (January 2012). *MapReduce and the Data Scientist.*

[37] Dr. ssa G. Lodi. *Collaborative Environment for Cyber Attacks Detection: The Cost of Preserving Privacy.* Pg. 13-14.

[38] J. Jablonski. *Introduction to Hadoop. A Dell Technical White Paper.* Pg. 03

[39] A. Thusoo (June 10, 2009). *Hive – A Petabyte Scale Data Warehouse using Hadoop.* http://www.facebook.com/note.php?note_id=89508453919

[40] *Cloudera. Introduction to Hive.*