Query Execution and Effect of Compression on NoSQL Column Oriented Data-store Using Hadoop and HBase

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By

Priyanka Raichand

(Roll No. 801132020)

Under the supervision of:

Dr. Rinkle Rani

Assistant Professor, CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

July 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Query Execution and Effect of Compression on NoSQL Column Oriented Data-store Using Hadoop and HBase"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Rinkle Rani** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Priyanka Raichand)

801132020

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

15/07/13

(Dr. Rinkle Rani)

Assistant Professor

Computer Science and Engineering Department

Thapar University

Patiala

Countersigned by

(Dr. Maninder Singh)

Head

Computer Science and Engineering Department

Thapar University

Patiala

(Dr. S. K Mohapatra)

Dean (Academic Affairs)

Thapar University

Patiala

Dedicated To, Lord Krishna, The Supreme Personality of Godhead, who in His nectarean words speaks as follows:

यत्करोषि यदश्नासि यज्जुहोषि ददासि यत् ।
यत्तपस्यसि कौन्तेय तत्कुरुष्व मदर्पणम् ॥२७॥
शुभाशुभफलैरेवं मोक्ष्यसे कर्मबन्धनैः ।
सन्न्यासयोगयुक्तात्मा विमुक्तो मामुपैष्यसि ॥२८॥

"Whatever you do, whatever you eat, whatever you offer or give away, and whatever austerities you perform—do that, O son of Kunté, as an offering to Me
In this way you will be freed from bondage to work and its auspicious and inauspicious results. With your mind fixed on Me in this principle of renunciation, you will be liberated and come to Me."

Bhagavad-gitä 9.27-28

Acknowledgement

I express my sincere gratitude to **Dr. Rinkle Rani**, Assistant Professor, Department of Computer Science and Engineering, for her comments and suggestions during the research work. Her continuous supervision and support helped me to keep this thesis work on the right track and achieve this final research thesis. Her ingenuity and kindness has motivated me along the way till the completion of this work.

I am also thankful to **Dr. Maninder Singh**, Head, Department of Computer Science Engineering, for his worthy suggestions and guidance.

I am indebted to **Dr. S.K Mohapatra**, Dean, Academic Affairs for facilitating with the contemporary infrastructure to perform thesis work.

I want to acknowledge my family and friends for providing me ideas, confidence & support; when I really needed them.

My appreciation goes to all, who have directly or indirectly helped me in completing my thesis.

Above all, my deep gratitude is for "**The Lord**" who helped me at every step and acted cornucopia of strength, affection and compassion. Without His will & mercy, I would have not been successful in completing thesis work.

(Priyanka Raichand)

Abstract

Today everyone is connected over the Internet and look to find relevant results instantaneously. Terabytes of data is being generated everyday and is subsequently processed. Data size in data-stores has been ever increasing. In large scale and high concurrency applications using the traditional relational database to store and query dynamic user data have come out to be inadequate.

Increased networking and business demands directly increases the cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. Compression techniques have not been considerably used in traditional relational database systems. The exchange between time and space for compression is not much pleasing for relational databases. Column oriented data-stores has all values of a single column stored as a row followed by all values of the next column. Such an approach of storing records helps in data compression since values of the same column are of the similar type and may repeat. Storing data in columns introduce a number of possibilities for better performance by compression algorithms.

Intend of the thesis is to see the effect of compression on NoSQL column oriented data-store. To perform this work, HBase - a Hadoop database was chosen. It is one of the most prominent NoSQL column oriented datastore and is being used by big companies like Facebook. Effect of compression and analysis has been performed with three compression codecs, **Snappy, LZO and GZIP** using only human readable text data. Hadoop Map Reduce framework has been used for loading the bulk data. Performance evaluation on compressed and uncompressed tables has been done by executing queries using advanced HBase API.

Results shows that using compression in NoSQL column oriented data-store like HBase increases the performance of the system overall. Snappy performs consistently well in saving CPU and network and memory usage. Second runner up is LZO. Whereas GZIP is not optimal choice where speed and memory usage is main concern but can work perfectly well where size disk space is a constraint.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Applications are undergoing transition from the traditional enterprise infrastructures to cloud infrastructures. With the development of the Internet and cloud computing, there is demand for high performance when reading and writing and to store and process big data effectively. Planet size web applications like Google, eBay, Facebook, Amazon etc. are a relatively recent development in the realm of computing and technology, requiring large scale to support hundreds of millions of concurrent users. Facebook, for example, adds more than 15 TB of data into its Hadoop cluster every day and is subsequently processing it all [1]. These applications are distributed across multiple clusters. These clusters consist of hundreds of server nodes that are located in multiple, geographically dispersed data centers. Data sizes in data-stores have been ever increasing. In efficiently managing and analyzing unprecedented sheer amount of data, scalable database management system plays an important role. In large scale and high concurrency applications using the traditional relational database to store and query dynamic user data have come out to be inadequate. In comparison to RDBMSs, NoSQL databases are more powerful and attractive in addressing this challenge [2].

Increased networking and business demands directly increases the cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. Compression techniques have not been considerably used in traditional relational database systems. The exchange between time and space for compression is not much pleasing for relational databases. NoSQL datastores were developed to deal with large scale needs and storage capacity. NoSQL community describes the acronym as "Not Only SQL". HBase is one of the most prominent NoSQL column oriented datastore. It is Hadoop database. HBase is not a column-oriented database in the typical RDBMS sense, but it utilizes an on-disk column storage format because HBase stores data on disk in a column-oriented format [1]. Storing data in columns introduce a number of possibilities for better performance from compression algorithms. Column- oriented databases save the data by grouping in columns. Column values are stored consecutively on disk in contrast to row-

oriented approach of databases which store entire rows contiguously [3]. Column oriented data-stores has all values of a single column stored as a row followed by all values of the next column. Such way of storing records helps in data compression since values of the same column are of the similar type and may repeat.

Intend of the thesis is to see the effect of compression on NoSQL column oriented data-store. To perform this work, HBase - a Hadoop database was chosen. It is one of the most prominent NoSQL column oriented datastore and is being used by big companies like Facebook. Effect of compression and analysis has been performed with three compression codecs, **Snappy, LZO and GZIP** using only human readable text data. Hadoop Map Reduce framework has been used for loading the bulk data. Performance evaluation on compressed and uncompressed tables has been done by executing queries using advanced HBase API.

Results shows that using compression in NoSQL column oriented data-store like HBase increases the performance of the system overall. Snappy performs consistently well in saving CPU, network and memory usage. Second runner up is LZO. Whereas GZIP is not optimal choice where speed and memory usage is main concern but can work perfectly well where size disk space is a constraint.

1.1 Advent of NoSQL

Today everyone is connected over the Internet and look to find relevant results instantaneously. Internet-scale applications require large scale to support hundreds of millions of concurrent users. These applications are distributed across multiple clusters that consist of thousands of server nodes located in multiple, geographically distributed data centres. Applications which are hosted in such heavily distributed environments present a number of major engineering challenges [4].

There is lot of data out there. The high volume of data being stored today is bursting. In the year 2000, 800,000 petabytes (PB) of data was stored in the world. It is expected this number to reach 35 zettabytes (ZB) by 2020. Twitter alone generates more than 7 terabytes (TB) of data every day, Facebook 10 TB, and some enterprises generate terabytes of data every hour of every day of the year. Big Data is characterized by its volume, velocity, and variety as shown in Figure 1.1[5]. Now question that clicks the mind is reliable storage of application data and state, which is the vital requirement for applications of any size. Technologies like cloud computing, large scale and high concurrency applications, such as search engines and planet size

web applications like Facebook, eBay, Amazon, twitter etc require databases to be able to store and process big data effectively and also demand high performance when reading and writing. All these issues have posed challenges to traditional relational databases. The relational database has appeared to be inadequate to store and query dynamic user data. Relational database systems make sure that ACID properties are met. RDBMS data model allows handling a few thousand requests over the course of a day and utilizing a single database instance on a server. To handle even more load, planned partitioning schemes, replication, and clustering can scale out the system. However, maintaining consistency and availability becomes increasingly difficult as the distributed system continues to grow. Strong consistency across too many server nodes requires multi-phase commit strategies and synchronous replication; this may increase the latency of requests thus making it sensitive to failures**.**

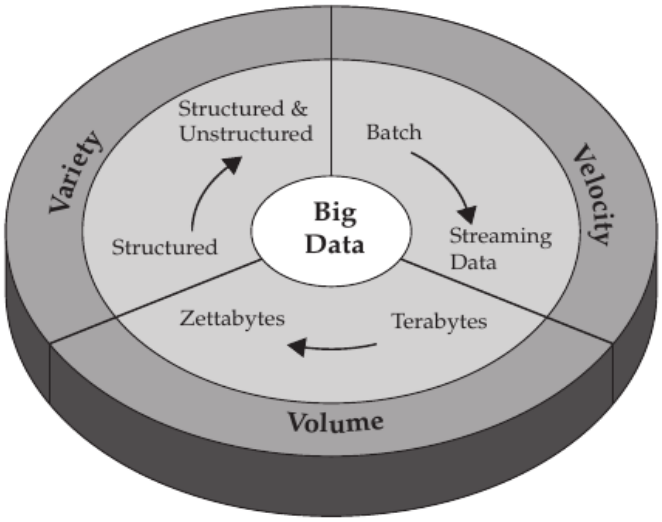


Figure 1.1: IBM characterizes Big Data as V^3 [5]

To conquer all these new challenges of Big Data NoSQL databases soon emerged. This new breed of non-relational distributed data-stores provides a simpler data model and focus on availability and incremental scalability, sometimes at the cost of consistency. NoSQL is an umbrella term for all databases and data stores that don't follow RDBMS principles. NoSQL data-stores follow **CAP theorem** that states a distributed system can't provide all three properties namely consistency, availability and partition tolerance simultaneously [6]. Architectures of NoSQL data-stores have been developed with keeping challenges in mind and to provide incremental scalability, high availability, and high write throughput.

NoSQL data-stores can be divided into four different categories [7].

- Key-value databases
- Column family databases
- Document databases
- Graph databases

1.2 NoSQL Column Oriented Data-Stores

Column-oriented databases save the data grouped by columns. Column family databases have rows and columns but storage on disk is organized such that columns of related data are grouped together in the same file. Such an approach allows accessing intended attributes without reading all of the other columns in the row. Column Family Stores are also known by other names as Column Oriented Stores, Extensible Record Stores and Wide Columnar Stores. Stores concept has been inspired by Googles Bigtable [8]. The data model is described as "sparse, distributed, persistent multidimensional sorted map" [8]. Feature of the model is that an arbitrary number of key value pairs can be stored within rows; since values cannot be interpreted by the system, Hence relationships between datasets and any other data types than strings are not supported natively. It provides both better performance and consistency by facilitating storage in chronological order of multiple versions of a value. For efficient data organization and partitioning; columns can be effectively grouped to column families. It permits the flexibility of columns and rows addition while runtime but often requires predefined knowledge of column families, which leads to less flexibility than key value stores and document stores offer. Widely known examples of Bigtable implementations are open source HBase [9] and Hypertable [10], whereas Cassandra [11] differs from the data model described above, since another dimension called super columns gets added. These super columns can contain multiple columns and can be stored within column families. Cassandra is more suitable to handle complex and expressive data structures. Due to their tablet format, column family stores have a similar graphical representation compared to relational databases. Main difference lies in their handling of null values. Considering a case with different kinds of attributes, relational databases would store a null value in each column a dataset has no value for. In contrast, column family stores only store a key value pair in one row, if a dataset needs it. This is what Google calls "sparse"

and which makes column family stores very efficient in domains with huge amounts of data with varying numbers of attributes. The data model of column family stores is more suitable for applications dealing with huge amounts of data stored on very large clusters, because data model can be partitioned very efficiently. Facebook created a high-performance Cassandra to help power its website. The Apache Software Foundation developed HBase is a distributed, open source database inspired from Google's Big Table. Columns are grouped into column families. It is also Schema free means columns and rows can be added at runtime. Figure 1.2 depicts the difference in storage model of column store and row stores.

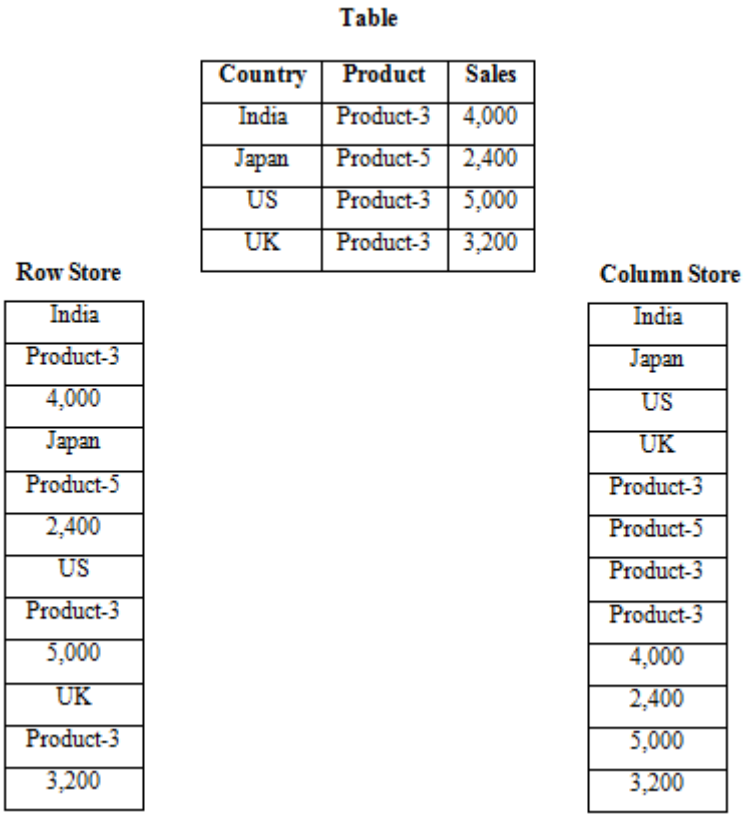


Figure 1.2: Storage Model of Column Store versus Row Store

1.3 Apache HBase

HBase is a distributed, persistent, strictly consistent, sparse, open source storage system. It is multidimensional sorted map which is indexed by rowkey, columnkey, and timestamp. HBase maintains maps of Keys to Values (key → value). Each of these mappings is called a KeyValue or a Cell. These cells are sorted by the key. It is quite important property as it allows for searching rather than just retrieving a value for a known key. Multidimensional means that the key itself has structure.

Each Key is consisting of four parts row-key, column family, column, and time-stamp. So the mapping takes place actually like (rowkey, column family, column, timestamp) →value. All data values in HBase are stored in form of bytes array. Features of HBase cover modular and linear scalability, consistent data access, automatic and configurable sharding of data. HBase tables can be accessed in two ways; one through an API and second they can act as input and output for MapReduce jobs run in Hadoop. One attractive feature of HBase is that it is distributed. Data can be spread over 100s or 1000s of machines and HBase manages the load balancing automatically. HBase does load shifting gracefully and transparently to the clients.

Working of HBase can be summarized in brief, HBase applications store data into tables which consist of rows and column families (containing columns); where, each row may have a different set of columns. Further, all columns are indexed with a user provided key column and are grouped into column families. Also, the table cell that is intersection of row and column coordinates is versioned and their content are uninterrupted array of bytes.

1.3.1 HBase Data Model

HBase models data is a little different from the relational systems. HBase data model is of semistructured shape since records can have different columns, variance in field size, and so on. HBase takes advantage of the semistructured shape of the data it stores. Advantage of semistructured shape is the loose coupling of data components which makes it easier for physical distribution. Basic constructs of HBase data model can be described as follows:

1. **Table:** Applications stores data into an HBase table. Tables are made of rows and columns. The intersection of row and column coordinates known as cells are versioned.

2. **Cell Value:** A {row, column, and version} tuple precisely specifies a cell in HBase. Number of cells can exist with same row and column but differing only in its version dimension. A version is a long integer. While searching or reading from the store file the most recent values are found first as version dimension is stored in decreasing order.

3. **Row Key:** Table row keys are byte array and are defined by the application. Rows are lexicographically sorted with the lowest order appearing first in a table. The rowkey also provides a logical grouping of cells. All table accesses are via the table row key, it is primary key [12].

4. **Columns & Column Families:** Columns families in HBase are group of columns. Columns are known as column qualifiers. Column family and column qualifier together makes column key. Physically, all column family members are stored together in the file system because tuning and storage specifications are done at the column family level.

1.3.1.1 Logical Model – A Big Sorted Map

HBase identify data in a cell [rowkey, column family, column qualifier, and version]. Figure 1.3 depicts the sorted map of maps logical model of HBase. In a logical way HBase organizes data as a nested map of maps. At bottom level, Cell is a map keyed on version with the stored data as the value. Then column family is a map keyed on column qualifier with the cell as the value and at the top, a table is a map keyed on rowkey to the column family. In Java it is described as: Map<RowKey, Map<ColumnFamily, Map<ColumnQualifier, Map<Version, Data>>>>.

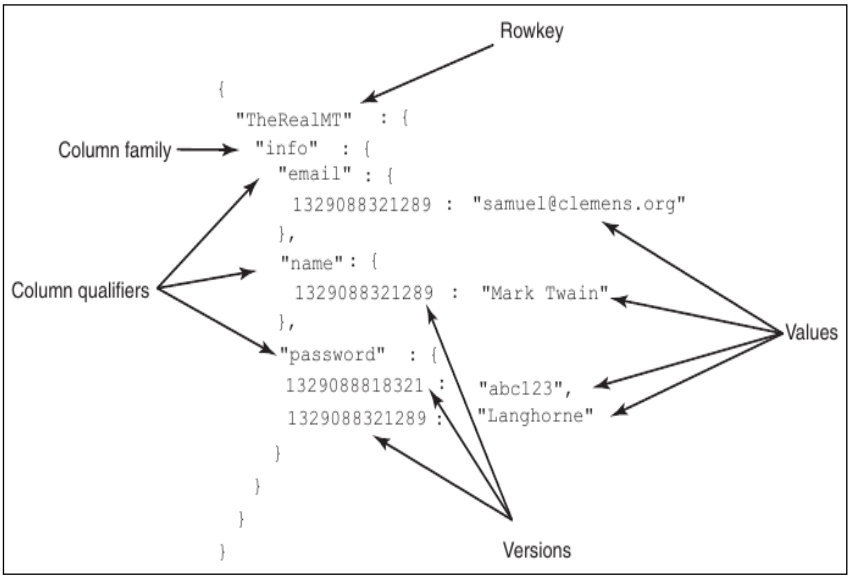


Figure 1.3: HBase Logical Data Model

1.3.1.2 Physical Model – Column Family Oriented

In logical model, grouping of columns as column families is expressed as a layer in the map of maps; whereas for physical modeling, each column family is assigned with set of HFiles on disk. Unique assignment provides physical isolation and allows easier management of one family to others as the HFiles for each column family are managed independently. Records in HBase are stored in the HFiles as key-value pairs. The HFile itself is a binary file and is not human-readable. An example has been shown in 1.4. Each record is a complete entry in the HFile. Notice that Mark's row consumes multiple records in the HFile. Each column qualifier and version gets its own record. Also, notice there are no unused or null records. HBase does not need to store anything to indicate the absence of data. Data from a single column family for a single row need not be stored in the same HFile.

"TheRealMT"	"info"	"email"	1329088321289	"samuel@clemens.org"
"TheRealMT"	"info"	"name"	1329088321289	"Mark Twain"
"TheRealMT"	"info"	"password"	1329088818321	"abc123"
"TheRealMT"	"info"	"password"	1329088321289	"Langhorne"

Figure 1.4: HBase Physical Data Model

1.4 HBase Architecture

A robust HBase architecture involves a few more parts than HBase alone [13]. It follows a master-worker pattern that means there are usually a master and a set of workers, known as range servers. Tables in HBase comprise rows and columns, but with a different kind of plan, it stores columns in a column-family together and can scale to billions of rows and millions of columns. Hence, size of each table can run into terabytes or even to petabytes. It is clear that such an enormous volume can not be hosted on a single machine.

1.4.1 Work behind the scenes

HBase runs on top of HDFS. Figure 1.5 depicts HBase architecture. Tables are horizontally split into regions, and regions are assigned to different region servers by the HBase master. RegionServers should be able to access HDFS. Regions are further vertically divided into stores by column families, and saved as store files in HDFS. Data replication in HDFS ensures high availability of table data in HBase.

During the runtime operations of the HBase, the ZooKeeper [14] is used to coordinate the activities of the master and region servers.

There exist two Catalog Tables in HBase architecture named -ROOT- and .META. that have got utmost importance. The .META. table keeps a list of all regions in the system where as -ROOT- keeps track of where the .META. table is. The information about the location of all regions and region servers is maintained with the help of these two tables. META. file keeps records for a user-space table, that is, the table that holds the data.

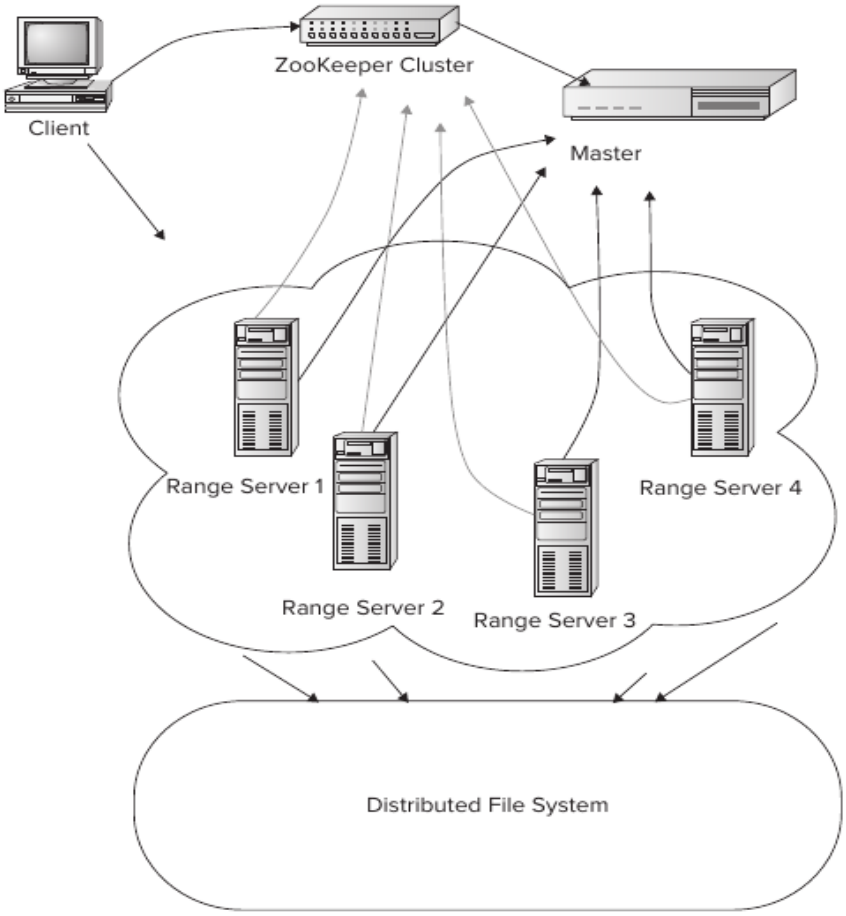


Figure 1.5: HBase Architecture [13]

1.4.2 Region Servers and Region

Each region maintains a separate store for each column-family in every table. Each store in turn maps to a physical file that is stored in the underlying distributed file system. For each store, HBase abstracts access to the underlying file system with the help of a thin wrapper that acts as the intermediary between the store and the underlying physical file. Each region has an in-memory store, or cache, and a

write-ahead-log (WAL) [13]. When data is written to a region, it is first written to the write-ahead-log, if enabled. Soon afterwards, it is written to the region's in-memory store. If the in-memory store is full, data is flushed to disk and persisted in the underlying distributed storage. Figure 1.6 recaps the core aspects of a region server and a region.

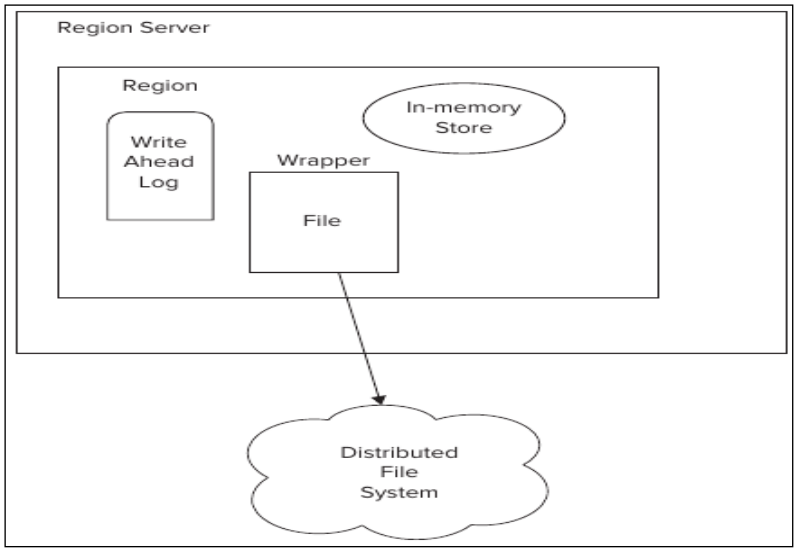


Figure 1.6: Region and Region Server

1.4.3 Distributed File System

For a distributed file system like the Hadoop distributed file system (HDFS) [15] master-worker pattern extends to underlying storage scheme also. Name node and a set of data nodes form a data base structure which is analogous to the configuration of master and range servers of column databases like HBase. Hence each physical storage file for an HBase column-family store ends up residing in an HDFS data node.

1.4.4 Zookeeper

HBase cluster also leverages an external configuration and coordination utility. This configuration program is known as Zookeeper. It keeps track of region servers, where the root region is hosted. It enables HBase to remove critical heartbeat messages that are required to be sent between the master and the region servers [1]. To access HBase the first time, a client accesses two catalogs via ZooKeeper. When a client wants to access a specific row it first asks ZooKeeper for the -ROOT- catalog. The -ROOT- catalog locates the .META. catalog relevant for the row, which in turn provides all the region details for accessing the specific row.

1.5 Compression

When dealing with the large volumes of data, the idea of compression is important. Enhancing networking and business demands directly increases the cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. It has two major benefits: first, it reduces the space needed to store files and second, it speeds up data transfer rate across the network, or to or from disk; which are like boon when dealing with large volume of data. For example Hadoop workloads are generally data-intensive, thus making the data reads a bottleneck in overall application performance. But by using some sort of compression of data faster reads are achievable. It is simply trading I/O load for CPU load as it requires uncompress also while reading data that would sacrifice few CPU cycles. Also if the infrastructure lack on disk capacity and has no problems in performance it becomes logical to use an algorithm that gives huge compression ratios. Large volume of disks are very cheaper than fast storage solutions so it is better that compression algorithm must be faster than being able to give higher compression ratios.

Compression techniques have not been considerably used in traditional relational database systems. The exchange between time and space for compression is not much pleasing for relational databases. Whereas storing data in columns introduce a number of possibilities for better performance from compression algorithms. Column- oriented databases save the data by grouping in columns. In a column oriented database, compression schemes encode multiple values at once. In a row-oriented database, such scheme does not bring good results because an attribute is stored as a part of an entire tuple [16].

There are two important things to see before choosing a compression scheme: first, splittable compression and second the compression and decompression speeds of the compression algorithm that is to be selected. In Hadoop, files are split (divided) if they are larger than the cluster's block size setting (normally one split for each block). For uncompressed files, this means individual file splits can be processed in parallel by different mappers [5].

When files, especially text files, are compressed, complications arise. For most compression algorithms, individual file splits cannot be decompressed independently from the other splits from the same file. More specifically, these compression algorithms are not splittable. For unsplittable compressed text files, MapReduce processing is limited to a single mapper. When file is compressed, it is no longer possible to parallelize the processing for each of the compressed file splits, because the file can be decompressed only as a whole, and not as individual parts based on the splits [5].

1.5.1 Compression in HBase

HBase has near-optimal write and brilliant read performance, and it uses the disk space efficiently by supporting pluggable compression algorithms that can be selected based on the nature of the data in specific column families. In HBase the data is stored in store files, called Hfiles. Hfiles can be compressed and are stored on HDFS. It saves disk I/O and instead pays with higher CPU utilization for compression and decompression while writing/reading data.

All compression algorithms exhibit a space/time trade-off means faster compression and decompression speeds usually come at the expense of smaller space savings [17]. Compression is defined as part of the table definition, enabled at the column family level, which is given at the time of table creation or at the time of a schema change. It can be outlined that why compression is important:

1. Compression reduces the number of bytes written to/read from HDFS.
2. Saves disk usage.
3. Improves the efficiency of network bandwidth when getting data from a remote server.

1.5.2 Compression Codecs in HBase

Various compression codecs are available to be used with HBase, mainly LZO, Snappy and GZIP. Following is the brief introduction of all the three codecs. Table 1.1 shows compression algorithm comparison in as Google published in 2005.

Table1.1: Comparison of Compression Algorithms by Google Inc

Algorithm	% remaining	Encoding	Decoding
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Snappy	22.2%	172 MB/s	409 MB/s

1. **LZO:** Lempel-Ziv-Oberhumer (LZO) [18] is a lossless data compression algorithm. It is focused on fast data decompression and low CPU usage, and written in ANSI C. HBase is not shiped with LZO because of licensing issues, HBase uses the Apache License, while LZO is using the incompatible GNU General Public License (GPL).By adding LZO compression support, HBase StoreFiles (Hfiles) uses LZO compression on blocks as they are written. HBase uses the native LZO library to perform the compression, while the native library is loaded by HBase via the hadoop-LZO Java library. Hadoop-LZO library brings splittable LZO compression to Hadoop [19].
2. **Snappy:** Snappy [20] is released by Google under the BSD License, provides the same compression used by Bigtable (Zippy). It behaves perfectly to provide high speeds and reasonable compression. The code is written in C++. Its aim is not to provide maximum compression, or compatibility with any other compression library; instead, it aims at providing very high speeds and reasonable compression. Snappy encoding is not bit-oriented, but byte-oriented. The first bytes of the stream are the length of uncompressed data, stored as a little-endian variant, which allows for variable-length encoding. The lower seven bits of each byte are used for data.
3. **GZIP:** The GZIP [21] compression algorithm compresses better than Snappy or LZO, but is slower in comparison. It comes with an additional savings in storage space. It comes shipped with HBase. GZIP is based on the DEFLATE algorithm, which is a combination of LZ77and Huffman coding. GZIP compression works by finding similar strings within a text file, and replacing those strings temporarily to make the overall file size smaller.

1.6 Motivation and Objectives

Motivation

Increased networking and business demands has directly increased the cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. Storing data in columns introduce a number of possibilities for better performance from compression algorithms. Column- oriented databases save the data by grouping in columns as column oriented data-stores has all values of a single column stored as a row followed by all values of the next column. Such way of storing records helps in data compression since values of the same column are of the similar type and may repeat.

Objectives

Following are the objectives of this thesis work:

1. Installation and hand-on experience on Hadoop, Map Reduce and HBase.
2. To analyze the effect of compression on performance of query execution.
3. To analyze the effect of compression on utilization of resources like CPU, memory, disk space and network, in case of NoSQL column oriented data-stores.

1.7 Thesis Organization

This thesis has been organized into 6 chapters, starting with this chapter,

Chapter 1 An introductory chapter; about to be covered.

Chapter 2 focuses on the research work related to compression in column oriented databases.

Chapter 3 states about the problem.

Chapter 4 concentrates on the detailed presentation of the implementation part for solving the stated problem.

Chapter 5 focuses on the acquired results and analysis.

Chapter 6 summarizes our findings, draws overall conclusions and suggests areas of future extension.

Chapter 2

Literature Review

2.1 Motivation for Column Oriented Datastores

Incremental scalability is a desirable characteristic of distributed systems. It makes the overall system more predictable, which in turn makes effective data center and capacity planning much easier. It can also provide sites with the flexibility of coping with big swings in load without having to redesign the entire system topology. Keeping the per-node administration overhead down is also critical. All of these benefits directly affect the costs of maintaining operations, which at Internet-scale, can be quite significant.

In its simplest form, incremental scalability means that when a system administrator adds a node to the cluster the overall capacity of the cluster will increase by a constant amount. This is why incremental scalability is also sometimes referred to as linear scalability. In order to achieve this, datastores must be able to dynamically partition the data over a set of nodes in the cluster [16]. The datastore must also be capable of self-balancing the data across its nodes, which in turn will theoretically lead to an even distribution of load.

Relational database operations, such as joins, are I/O bound operations and directly benefit from robust storage subsystems. Column-oriented datastores do not support joins and they also buffer large amounts of data in memory. In general, reads within a column-oriented datastore are more CPU bound rather than I/O bound. Writes within a column-oriented datastore first occur against an in-memory data structure that is periodically flushed to disk. This results in sequential I/O for chunks of data, which is less I/O intensive than the many random writes of smaller objects that occur in relational databases [16].

It is because of these characteristics that column-oriented datastores can make use of storage subsystems consisting of commodity hardware and still remain preferment. A node in such a system may only need 2 or 3 consumer grade (commodity) hard drives to sufficiently handle its storage needs. Contrast that with the needs of a node within a relational database cluster, where 15k RPM hard drives with fiber channel interfaces and expensive RAID controllers are the norm. Multiplying the cost difference times the number of nodes needed to effectively

support operations yields a significant savings in overall cost. Column-oriented datastores excel at workloads that favor availability and low latency over strong consistency [16]. Examples of such workloads include persisting a user's session, managing a shopping cart, or providing storage for server log files. Since the per-node cost within the cluster is low, a high level of redundancy can be achieved in much more affordable way.

2.2 Basic Properties of Column Oriented Datastores

In highly replicated distributed systems, there is often a great emphasis on the durability of data and not just persistence. Durability requires that writes from a particular operation are stored in such a way that if an exception were to occur, that data can be recovered. Copies might be stored in memory or on disk, but they are typically written to a commit log. Durability can be achieved through replication as well, while persistence is focused on storing the data on disk for long term storage. Some column-oriented datastores, such as Dynamo, support pluggable storage engine architecture for persistence [16].

Querying within a column-oriented datastore is often limited to key only lookups, which are provided by each datastore's own API. There is no query language, so data access is totally programmatic. Datastores provide namespaces, such as column families and keyspaces. These must be specified when querying data. Versioning techniques are critical to the concurrency model of column-oriented datastores. Updates within a row are commonly implemented as atomic operations with a timestamp used to denote the version. In some cases, the latest timestamp is the true version. Security and access control is not a strong focus with column-oriented databases. This is an area where relational databases are much more robust. In general, most popular column-oriented datastores place a lesser value on consistency and integrity compared to fault tolerance and low latency response. The alternative consistency model that these datastores focus on is called eventual consistency. To achieve high availability, replication amongst nodes is utilized extensively. Another wrinkle with respect to the integrity of column-oriented datastores is the fact that there is little or no support for types. Values are stored as uninterpreted byte strings, so it is up client applications in order to maintain consistent typing of values. Support for recovery is effectively handled by using a commit log. Write operations are written to the commit log after finishing successfully.

2.3 The Rise of HBase

HBase was created in 2007 at Powerset and was initially, part of the contributions in Hadoop. HBase is the **H**adoop data**Base**. It is a distributed, scalable Big datastore. HBase is used when requirement is of random, real-time read/write access to Big Data. The goal of the HBase project is to host very large tables, billions of rows multiplied by millions of columns on clusters built with commodity hardware. HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable. Just as Bigtable leverages the distributed data storage provided by the Google File System, HBase provides Bigtable-like capabilities on top of Hadoop and HDFS [1].

Fundamentally, it is a platform for storing and retrieving data with random access, that is data can be written in a way as one likes and can be read back again as one needs. HBase stores structured and semistructured data naturally so can easily load it with tweets, parsed log files and catalog of all products right along with the customer reviews. It can store unstructured data too, as long as it is not too large. It does not care about types and allows for a dynamic and flexible data model that does not constrain the kind of data that is being stored [4].

The column-oriented architecture allows for huge, wide, sparse tables as storing NULLs is free. Because each row is served by exactly one server, HBase is strongly consistent, and using its multiversioning helps in avoiding edit conflicts caused by concurrent decoupled processes or retain a history of changes. HBase is a distributed, persistent, strictly consistent storage system with near-optimal write, in terms of I/O channel saturation and excellent read performance, and it makes efficient use of disk space by supporting pluggable compression algorithms that can be selected based on the nature of the data in specific column families [1].

HBase extends the Bigtable model, which only considers a single index, similar to a primary key in the RDBMS world, offering the server-side hooks to implement flexible secondary index solutions. In addition, it provides push-down predicates, that is, filters, reducing data transferred over the network. There is no declarative query language as part of the core implementation, and it has limited support for transactions. Row atomicity and read-modify-write operations make up for this in practice, as they cover most use cases and remove the wait or deadlock-related pauses experienced with other systems [1]**.** HBase handles shifting load and

failures gracefully and transparently to the clients. Scalability is built in, and clusters can be grown or shrunk while the system is in production. Changing the cluster does not involve any complicated rebalancing or resharding procedure, but is completely automated.

HBase proved to be a powerful tool, especially in places where Hadoop was already in use. Even in its infancy, it quickly found production deployment and developer support from other companies. Today, HBase is a top-level Apache project with thriving developer and user communities. It has become a core infrastructure component and is being run in production at scale worldwide in companies like StumbleUpon, Trend Micro, Facebook, Twitter, Salesforce, and Adobe [1].

2.4 Compression in Column Oriented Datastores

Research in database compression has been approximately for nearly a century though compression techniques were not frequently used until 1990's [3]. However, it was not until the 1990s that research first concentrated on how compression affects database performance [22]-[26]. Most work concluded that there were a limited number of compression algorithms that could be used as the CPU cost of decompression outweigh the I/O savings. Therefore, most systems concentrated on the use of light-weight compression techniques. Light-weight schemes sacrifice compression ratio in order to ensure low compression and decompression CPU requirements.

It is only today that these are being put to use inside important information systems. It may be so because earlier most of the focus was on cutting down the size of the stored data, but in 90's researchers started to concentrate on affects of compression on databases performance [22]-[25]. Few researchers have investigated the effect of compression on database systems and their performance [26]-[28]. This research work has discovered that compression does reduce I/O cost but if the cost of compressing/decompressing outbalances this saved cost, then it results in reduced overall performance of the database. This trade-off becomes more favorable for compression with the improvements in CPU speed [29]. Many researchers have chew over text compression schemes such as Huffman encoding that is based on letter frequency. String matching schemes are being looked by more recent research in compression [30]-[33].

Following the initial studies, research turned to reducing the cost of decompression. In early systems, data was decompressed immediately after it was read from disk. Few Researchers [17] suggested that it was not always necessary to decompress data. In particular, it was possible to lazily decompress the data, if and when that data was used by the executor. In such systems, data is compressed on the attribute level and held compressed in memory. Data would only be decompressed when an operator were to access the data.

Along with these traditional techniques, column oriented databases are also substantially best for compression schemes that compress data from more than one row at a time thus allowing many great kind of workable compression algorithms. For example, for compressing sorted data in a column oriented database, run-length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is a pleasing technique [3].

Generally higher compression ratios in column oriented datastores is observed because consecutive values of the same column are of the similar type and may repeat, whereas adjacent attributes in a tuple are not [34]. The overhead incurred by CPU for iterating through a page of column values (particularly when all column values are the same size) tends to be less. Column oriented databases can store different columns in unlike sort-orders, further enhancing the possibility for compression [34]. Column oriented compression techniques also improve CPU performance by allowing operations directly on compressed data. This is particularly true for compression schemes like run length encoding that refer to multiple entries with the same value in a single record [3]. Following is the description of various compression schemes for column oriented databases. For each scheme, a brief description of the traditional version of the scheme as previously used in row oriented database systems and later in regard with column oriented databases is given [17].

A. Dictionary Encoding: Today's database systems perhaps dictionary encoding schemes are most dominant types of compression schemes. These schemes replace frequent patterns with smaller codes. A column-optimized variant of dictionary encoding is new implementation. Row oriented datastores are basically not capable of blending attributes from more than one tuple in a single entry thus making dictionary encoding schemes not to function fully as they can only map attribute values from a single tuple to dictionary entries.

1. **Dictionary Encoding Algorithm:** Dictionary encoding algorithm [17] calculates the number of bits, X, needed to encode a single attribute of the column (which can be calculated directly from the number of unique values of the attribute). It then calculates how many of these X-bit encoded values can fit in 1, 2, 3, or 4 bytes. For example,[17] if an attribute has 32 values, it can be encoded in 5 bits, so 1 of these values can fit in 1 byte, 3 in 2 bytes, 4 in 3 bytes, or 6 in 4 bytes. Suppose that the 3-value/2-byte option was chosen. In that case, a mapping is created between every possible set of 3 5-bit values and the original 3 values. For example, if the value 1 is encoded by the 5 bits: 00000; the value 25 is encoded by the 5 bits: 00001 and the value 31 is encoded by the 5 bits 00010; then the dictionary would have the entry (read entries right-to-left)

   $$X000000000100010 \rightarrow 31\ 25\ 1$$

   Where X indicates an unused "wasted" bit. The decoding algorithm for this example is then straightforward: read in 2-bytes and lookup entry in dictionary to get 3 values back at once. Column oriented databases are quite I/O efficient that queries on the column become CPU limited after applying even small amount of compression[37]. So the I/O savings that one get by not wasting the extra space is unimportant. Thus, it is worth byte-aligning dictionary entries to obtain even modest CPU savings [3].

2. **Cache-Conscious Optimization [17][36]:** The decision as to whether values should be packed into 1, 2, 3, or 4 bytes is decided by requiring the dictionary to fit in the L2 cache. In the above example, each entry takes 2 bytes and the number of dictionary entries is $32^3 = 32768$. Therefore the size of the dictionary is 393216 bytes which is less than half of the L2 cache (1MB).

B. Run Length Encoding: Run Length Encoding (RLE) is a simple and popular data compression algorithm. Run-length encoding compresses [3] based on the idea of replacing the same long sequence in a column to a compact singular representation. Thus, it is well-suited for columns that are sorted or that have reasonable-sized runs of the same value. These runs are replaced with triples: (value, start position, run length) where each element of the triple is given a fixed number of bits. In row-oriented systems, RLE is only used for large

string attributes that have many blanks or repeated characters. But in column oriented RLE can be much more greatly used systems where attributes are stored consecutively and runs of the same value are common (mainly in columns that have less distinct values).

C. **Null Suppression:** Null compression scheme has many variants but the basic logic is to replace consecutive zeros or blanks in the data are deleted and replaced with a description of how many there were and where they existed [3]**.** Naturally, this technique works great on data sets where zeros or blanks appear frequently. Variable field sizes are encoded in the number of bytes needed to store each field in a field prefix. This allows to exclude heading nulls needed to pad the data to a fixed size. For example, [3] for integer types, rather than using the full 4 bytes to store the integer, encoding is done for exact number of bytes needed using two bits (1, 2, 3, or4 bytes) and placing these two bits as prefix of the integer.

D. **Lempel-Ziv Encoding:** The Lempel-Ziv compression algorithms were developed in 1977-78. Lempel-Ziv [38, 39] compression is the most widely used technique for lossless file compression. The UNIX command GZIP is based upon this algorithm only. Lempel-Ziv replaces variable sized patterns with fixed length codes unlike to Huffman encoding which produces variable sized codes. In Lempel-Ziv encoding knowledge about pattern frequencies in advance is not a requirement as it builds the pattern table dynamically as it encodes the data. The main idea is to analyze the input sequence into non-overlapping blocks of different lengths and constructing a dictionary of blocks seen thus far. Later on occurrences of these blocks are replaced by a pointer to an earlier occurrence of the same block.

Chapter 3

Problem Statement

With the development of the Internet and cloud computing, there is demand for high performance when reading and writing and to store and process big data effectively. Planet size web applications like Google, eBay, Facebook, Amazon etc. are a relatively recent development in the realm of computing and technology, requiring large scale to support hundreds of millions of concurrent users. In large scale and high concurrency applications using the traditional relational database to store and query dynamic user data have come out to be inadequate. NoSQL datastores were developed to deal with large scale needs and storage capacity.

Increased networking and business demands has directly increased the cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. Storing data in columns introduce a number of possibilities for better performance from compression algorithms. Column- oriented databases save the data by grouping in columns as column oriented data-stores has all values of a single column stored as a row followed by all values of the next column. Such way of storing records helps in data compression since values of the same column are of the similar type and may repeat.

HBase is one of the most prominent NoSQL column oriented datastore. HBase is all capable to face these new challenges as it allows horizontal scalability, support high-availability, support pluggable compression codecs and have the flexibility to handle semi-structured data.

Intend of the thesis is to see the effect of compression on NoSQL column oriented data-store. To perform this work, HBase - a Hadoop database is chosen. It is one of the most prominent NoSQL column oriented datastore and is being used by big companies like Facebook. It aims at performance analysis of query execution using three compression codecs, Snappy, LZO and GZIP using only human readable text data. It is important to know how compression in NoSQL column oriented databases helps in efficiently utilizing the available resources mainly CPU, memory, network and disk.

Chapter 4

Implementation

4.1 Installation and Configuration of Hadoop and HBase

HBase runs on the top of Hadoop and uses HDFS to store all files. These files are divided into blocks when stored within HDFS. Compression analysis journey was started with installation and configuration of Hadoop-1.0.4 and HBase-0.94.5 in pseudo distributed mode on a single Linux box-64-bit (Kubuntu) with Intel corei5 processor, 3.84GB of RAM and 105GB disk space. Figure 4.1 is the snapshot of HDFS Namenode after all installation and configuration.

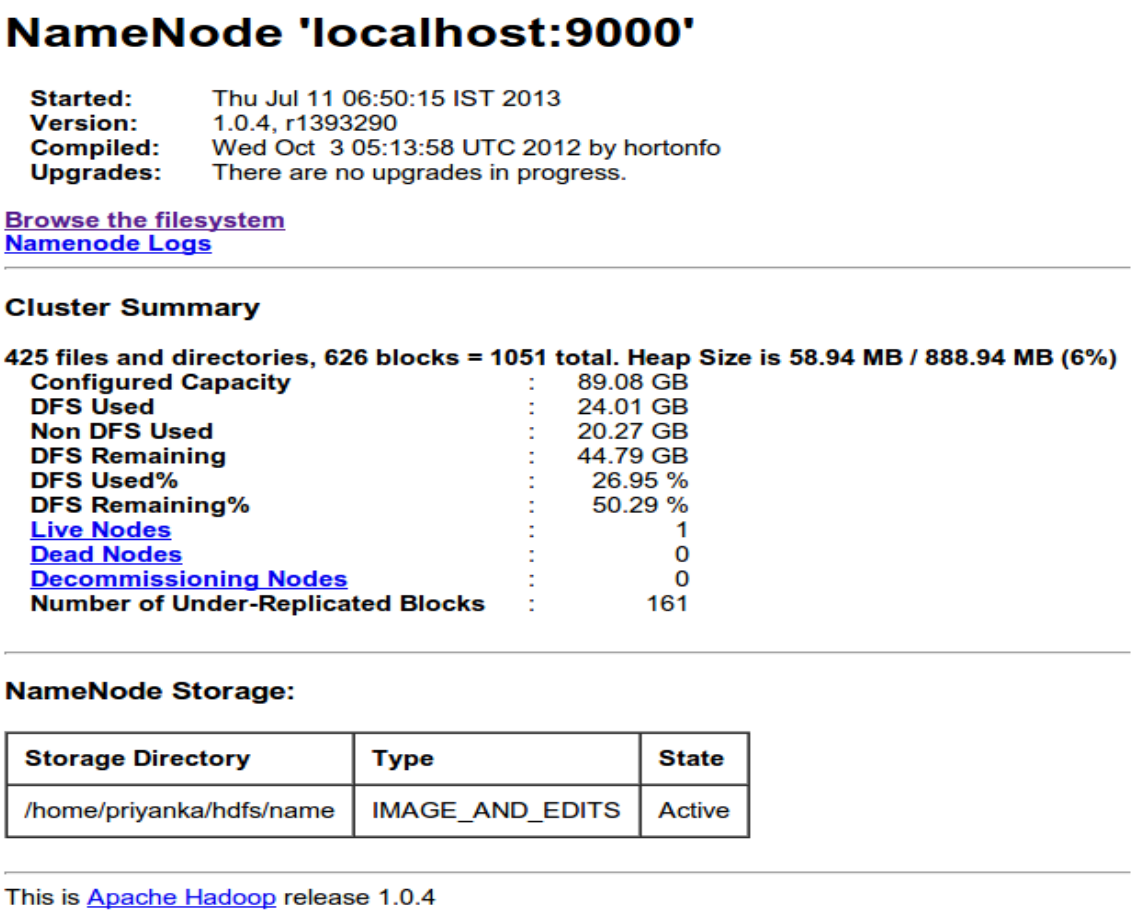


Figure 4.1: HDFS Namenode

Figure 4.2(a) depicts the contents in HDFS. All the text files are in user directory from where data is loaded in HBase. The directory HBase shows all directories inside HBase with the same name as the tables. Figure 4.2(b) depicts the same.

Contents of directory /

Goto : / go

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
hbase	dir				2013-07-10 12:52	rwxr-xr-x	priyanka	supergroup
home	dir				2013-07-10 07:06	rwxr-xr-x	priyanka	supergroup
system	dir				2013-07-09 00:41	rwxr-xr-x	priyanka	supergroup
user	dir				2013-06-29 15:50	rwxr-xr-x	priyanka	supergroup

Figure 4.2(a): Home Directory

Contents of directory /hbase

Goto : /hbase go

Go to parent directory

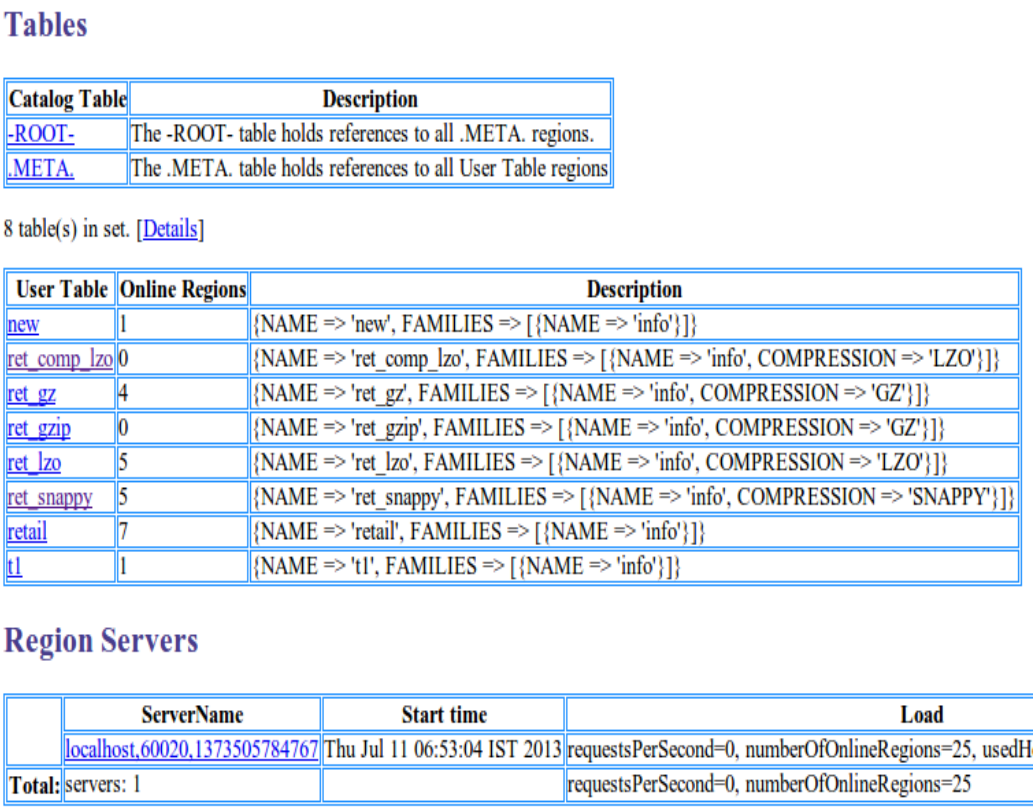
Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
-ROOT-	dir				2013-07-10 12:52	rwxr-xr-x	priyanka	supergroup
.META.	dir				2013-07-10 12:52	rwxr-xr-x	priyanka	supergroup
.archive	dir				2013-07-10 07:26	rwxr-xr-x	priyanka	supergroup
.corrupt	dir				2013-03-30 05:59	rwxr-xr-x	priyanka	supergroup
.logs	dir				2013-07-10 21:18	rwxr-xr-x	priyanka	supergroup
.oldlogs	dir				2013-07-10 12:52	rwxr-xr-x	priyanka	supergroup
hbase.id	file	0.04 KB	1	64 MB	2013-03-28 04:32	rw-r--r--	priyanka	supergroup
hbase.version	file	0 KB	1	64 MB	2013-03-28 04:32	rw-r--r--	priyanka	supergroup
new	dir				2013-06-27 11:28	rwxr-xr-x	priyanka	supergroup
ret_comp_lzo	dir				2013-06-15 02:35	rwxr-xr-x	priyanka	supergroup
ret_gz	dir				2013-07-08 23:35	rwxr-xr-x	priyanka	supergroup
ret_gzip	dir				2013-07-04 12:55	rwxr-xr-x	priyanka	supergroup
ret_lzo	dir				2013-06-17 22:23	rwxr-xr-x	priyanka	supergroup
ret_snappy	dir				2013-06-29 15:05	rwxr-xr-x	priyanka	supergroup
retail	dir				2013-06-21 21:51	rwxr-xr-x	priyanka	supergroup
t1	dir				2013-06-15 02:24	rwxr-xr-x	priyanka	supergroup

Figure 4.2(b): Tables Directories inside HBase Directory

After configuring Hadoop, HBase configuration is done so that HBase and Hadoop files system can be combined to store data. For configuring there is need to add some properties values in HBase-site.xml file in its conf directory. Figure 4.3 is a snapshot of HBase Master running at localhost:60010. HMaster performs low-level file operations. Displaying all tables present in HBase along with description. Initially there was only one table 'retail' with a column family 'info' and created and defined using HBase shell as follows:

HBase(main):001:0> create 'retail, 'info'

At this time compression is not enabled for retail table and table is empty. A table in a HBase must have at least one column family.



Tables

Catalog Table	Description
-ROOT-	The -ROOT- table holds references to all .META. regions.
.META.	The .META. table holds references to all User Table regions

8 table(s) in set. [Details]

User Table	Online Regions	Description
new	1	{NAME => 'new', FAMILIES => [{NAME => 'info'}]}
ret_comp_lzo	0	{NAME => 'ret_comp_lzo', FAMILIES => [{NAME => 'info', COMPRESSION => 'LZO'}]}
ret_gz	4	{NAME => 'ret_gz', FAMILIES => [{NAME => 'info', COMPRESSION => 'GZ'}]}
ret_gzip	0	{NAME => 'ret_gzip', FAMILIES => [{NAME => 'info', COMPRESSION => 'GZ'}]}
ret_lzo	5	{NAME => 'ret_lzo', FAMILIES => [{NAME => 'info', COMPRESSION => 'LZO'}]}
ret_snappy	5	{NAME => 'ret_snappy', FAMILIES => [{NAME => 'info', COMPRESSION => 'SNAPPY'}]}
retail	7	{NAME => 'retail', FAMILIES => [{NAME => 'info'}]}
t1	1	{NAME => 't1', FAMILIES => [{NAME => 'info'}]}

Region Servers

	ServerName	Start time	Load
	localhost,60020,1373505784767	Thu Jul 11 06:53:04 IST 2013	requestsPerSecond=0, numberOfOnlineRegions=25, usedH
Total:	servers: 1		requestsPerSecond=0, numberOfOnlineRegions=25

Figure 4.3: HMaster Web UI

4.2 Loading Bulk Data in HBase

For loading the bulk data from the file retail.txt stored in HDFS, Hadoop Map/Reduce framework is used. File size is 6.5GB. HBase tables are schema free, that is, many number of columns can be added or removed later on after the creation of table. Figure 4.4 shows schema of the all tables.

4.2.1 MapReduce Job for Loading Data

MapReduce process shows how the data is processed. The first thing that happens is the split, which is responsible for dividing the input data into reasonably sized chunks that are then processed by one server at a time. The MapReduce framework take care of all the underlying details regarding parallelization, data distribution and load balancing while the user is concerned only about the local computations executed on every machine. Figure 4.5 summarizes processing of data in a MapReduce job. These computations are divided into two categories: the map and the reduce computations.

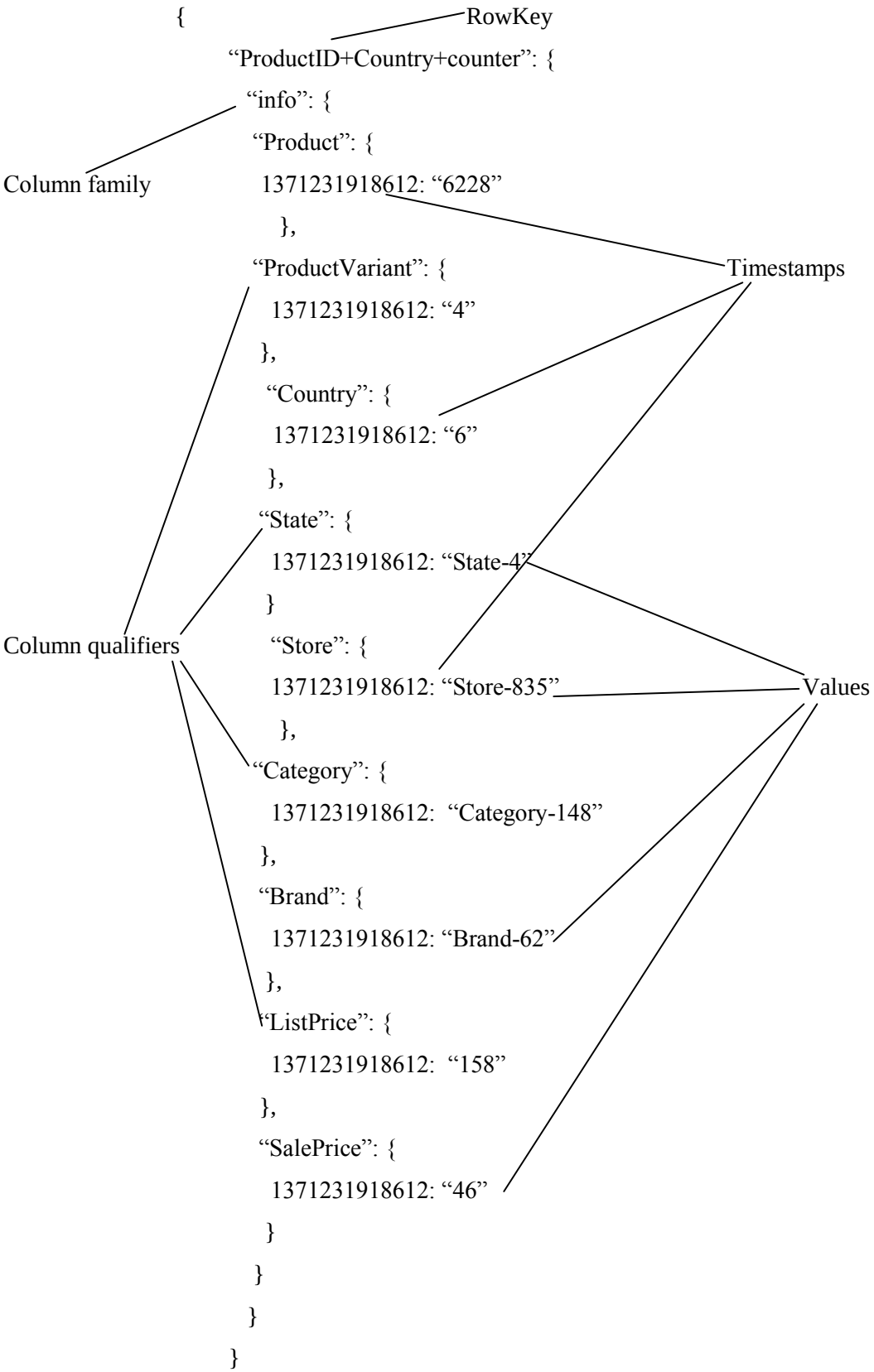


Figure 4.4: Table Schema [Logical View]

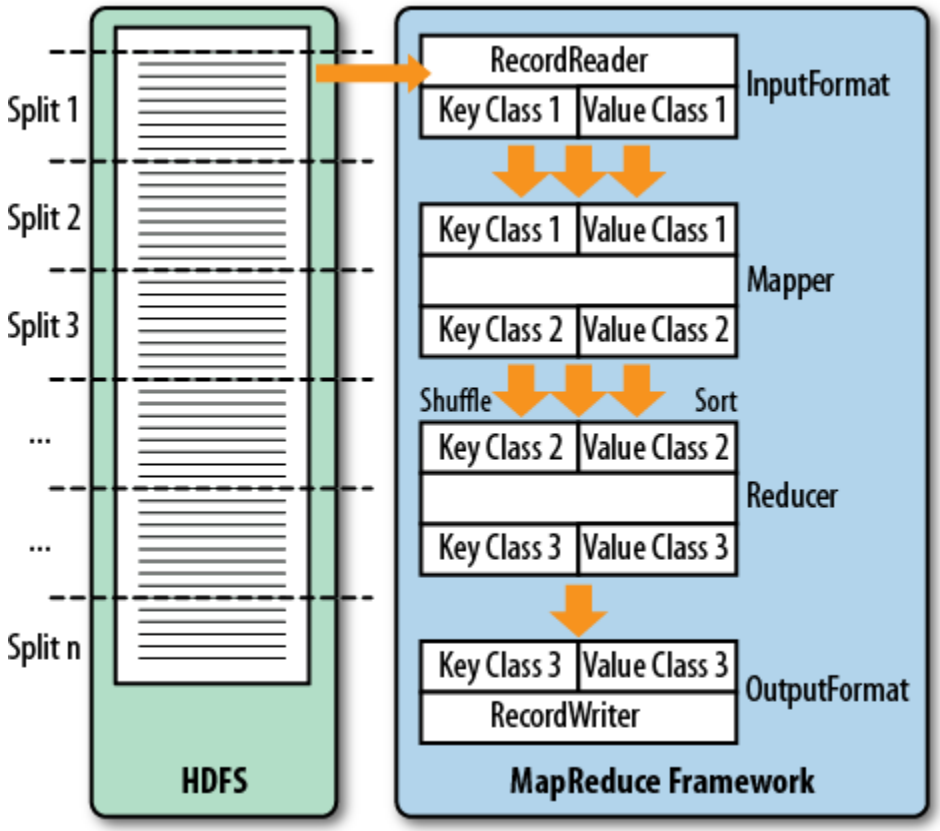


Figure 4.5: The MapReduce Process

Following is the pseudo code of Map/Reduce job for loading bulk data. It is implemented in JAVA using eclipse IDE. It contains mainly three classes - Mapper class, Reducer class and finally the Driver class. Row key design in HBase tables is important as it is the only means to access value in the cell. In this program row key is taken as the combination of three values, two columns namely product and country and a counter to make it more distinct as it act as the Primary key to access the values in cells.

Input: Texts file from HDFS

Main class name: public class RetailDataLoader **{**

Global variable: public static enum KEY_COUNTER;

Mapper class: public static class RetailMapper extends provided Hadoop class

Parameter passed: <record InputFormat, record OutputFormat> **{**

Return: key / value pair

Local variables: String line, String array parts, String fHalf, String sHalf, String k, Text outputKey;

Map sub method: public void map

Parameters passed: (key / value pair) **{**

Body: creates key / value pair

k = fHalf + "." + sHalf;

value = line;

Return: OutputFormat key / value pair

}}}

Reducer class: public static class RetailReducer extends **TableReducer**

Parameter passed: <output of Mapper method> **{**

Local variable: byte[] rowKey;

Reduce method: public void reduce **{**

Parameters passed: (key / value pair)

Body:

Local variable: Text v

for: each v in value **{**

increment global counter variable;

call method getRowKey;

store rowKey in the given table;

get data for columns;

store data in the columns in given table;

}}}

Driver class: public static void main(String[] args) throws IOException **{**

Body:

set Configuration;

add resources;

defines path of the text file from hdfs;

define job with required classes;

set file input format;

set jar by class;

set Mapper class;

set Reducer class;

set Map OutputKey class;

set Map OutputValue class;

set InputFormat class;

configure identity reducer to store the parsed data;

getRowKey: public static byte[] getRowKey **{**

Parameters passed: <mapper key>

Body:

Local Variables: String keys, String countryName

set rowKey value;

Return: rowKey

}}

Output

```
Starting MR Job...
13/06/15 08:34:23 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:zookeeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:host.name=localhost
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:java.version=1.6.0_45
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:java.vendor=Sun Microsystems Inc.
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:java.home=/usr/lib/jvm/java-6-oracle/jre
13/06/15 08:34:25 INFO zookeeper.ZooKeeper: Client environment:java.class.path=/home/priyanka/workspace/dataloader/bin:/home/priyanka/hbase-0.94.5/hbase-0.94.5.jar:/home/priyanka/hbase-0.94.5/lib/commons-configuration-1.6.jar:.....
13/06/15 08:34:25 INFO mapred.JobClient: Running job: job_201306150820_0001
13/06/15 08:34:26 INFO mapred.JobClient:  map 0% reduce 0%
13/06/15 08:34:42 INFO mapred.JobClient:  map 1% reduce 0%
13/06/15 08:34:45 INFO mapred.JobClient:  map 2% reduce 0%
13/06/15 08:37:12 INFO mapred.JobClient:  map 10% reduce 3%
13/06/15 08:37:21 INFO mapred.JobClient:  map 11% reduce 3%
13/06/15 08:37:42 INFO mapred.JobClient:  map 12% reduce 3%
13/06/15 08:38:00 INFO mapred.JobClient:  map 13% reduce 3%
13/06/15 08:38:12 INFO mapred.JobClient:  map 13% reduce 4%
13/06/15 08:38:18 INFO mapred.JobClient:  map 14% reduce 4%
13/06/15 08:38:39 INFO mapred.JobClient:  map 15% reduce 4%
….....
….....
13/06/15 10:16:28 INFO mapred.JobClient:  map 100% reduce 98%
13/06/15 10:18:38 INFO mapred.JobClient:  map 100% reduce 99%
13/06/15 10:22:59 INFO mapred.JobClient:  map 100% reduce 100%
13/06/15 10:23:22 INFO mapred.JobClient: Job complete: job_201306150820_0001
13/06/15 10:23:24 INFO mapred.JobClient: Counters: 29
13/06/15 10:23:24 INFO mapred.JobClient:     CPU time spent (ms)=5388780
13/06/15 10:23:24 INFO mapred.JobClient:     Reduce input records=80672928
13/06/15 10:23:24 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=39210479616
13/06/15 10:23:24 INFO mapred.JobClient:     Map output records=80672928
Data has been loaded. Time taken to load the data: 6540659 milliseconds.
```

To confirm and see the loaded data scan command (written below) using HBase shell is used.

HBase(main):003:0> scan 'retail'

Snippet of the output of table scan is shown below. This table has 80672928 rows and nine columns.

Output

```
\x00\x00\x00\x00\x00\x00\x00\x01.Country-1.\x00\x00\x00\x00\x0

column=info:product, timestamp=1312821497945, value=3
column=info:productVariant, timestamp=1312821497945, value=4
column=info:Country, timestamp=1312821497945, value=Country-6
column=info:State, timestamp=1312821497945, value=State-10
column=info:Store, timestamp=1312821497945, value=Store-34
column=info:Category, timestamp=1312821497945, value=Category-180
column=info:Brand, timestamp=1312821497945, value=Brand-23
column=info:ListPrice, timestamp=1312821497945, value=260
column=info:SalePrice, timestamp=1312821497945, value=230
.
.
.
80672928 row(s)
```

Next section explains the installation of the compression codecs for HBase, mainly Snappy and LZO.

4.3 Activating Compression Codecs in HBase

Activating compression Snappy and LZO codecs in HBase it is required to build hadoop-lzo and native libraries and hadoop-snappy and native libraries from the source and configuring Hadoop and HBase to use these libraries. These were installed under the $HBase_HOME/lib and $HBase_HOME/lib/native directories, respectively. HBase also supports GZIP codec and that is shipped with it does not needs to be installed.

4.3.1 Activating LZO/Snappy

This section outlines the steps needed to activate LZO/Snappy compression in HBase.

Step 1. Installation of Apache Ant, Maven, libtoolize.

Step 2. Get the latest hadoop-lzo/ hadoop-snappy.

Step 3. Build hadoop-lzo / hadoop-snappy from source.

Step 4. Build the native and Java hadoop-lzo/ hadoop-snappy libraries from source, depending on OS.

Step 5. This step creates the hadoop-lzo/ hadoop-snappy build/native directory and the hadoop-lzo/ build/hadoop-lzo-1.0.4.jar file.

Step 6. Copy the built libraries to the $home/priyanka/hbase-0.94.5/lib and $home/priyanka/hbase-0.94.5/lib/native directories on master node.

Step 7. Add the configuration of hbase.regionserver.codecs to hbase-site.xml file.

Once the installation is complete, verification is done by using compression test tool mechanism available in HBase. HBase ships with a tool, given below, to test whether compression is set up properly.

**Hbase class org.apache.hadoop.HBase.util.CompressionTest **
hdfs://localhost:9000/user/priyanka/test.txt snappy

The tool reports SUCCESS, and therefore confirms that this compression type for a column family definition can be used. For using compression algorithms, it is to be added at the time of column family creation in table definition. Next the defining of three new tables named 'ret_LZO', 'ret_snappy' and 'ret_gz' using HBase shell is done. For example:

Hbase(main):001:0> create 'ret_LZO', { FAMILY => 'info', COMPRESSION => 'LZO'}

By adding compression support in HBase table at column family level, HBase StoreFiles (HFiles) can now use compression on blocks as they are written. For performing compression HBase uses the native library like of LZO and Snappy. The native library is loaded by HBase via the hadoop-lzo / hadoop-snappy library.

Next section implements queries on these tables, on the basis of which performance evaluation of compression codecs is accomplished.

4.4 Implementation of Queries in HBase

To further evaluate the effect of compression and improved performance in terms of CPU and network utilization, disk space and memory used, query implementation is done. Compressed tables and the uncompressed one are queried for the analysis purpose. Queries have been implemented in JAVA and using advanced HBase API mainly filters. Eclipse IDE is used to compile and run these queries. Following is the pseudo code representation of queries followed by output snapshots.

Query 1. To find countries where the given product has been sold

```
Class Definition: public class FirstQuery {
Body:
        Main method: public static void main(String[] args) {
         create the required configuration;
         instantiate new table reference;
         call firstQuery method;        }
firstQuery method: private static void firstQuery {
Parameter passed: HTable client;
Local variable: long productID, Set<String> set;
Body:
        create RowFilter;
        create an empty Scan instance;
        pass filter to scan;
        get a scanner to iterate over the rows;
         for (Result r : resultScanner) {
                add countries name in object set;
         }
        for (String s : set) {
                print countries;
        close resultScanner; }
 }}
```

Output:

```
Processing Query 1...
Countries where 'Product 3925' has been sold are :
Country-1
Country-2
Country-3
Country-4
Country-5
Country-6
Country-7
Country-8
Country-9
Elapsed Time For Query 1 : 131821
```

Query 2. Count of Products in the given country

Class Definition: public class SecondQuery **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

call secondQuery method; **}**

secondQuery method: private static void secondQuery **{**

Parameter passed: HTable client;

Local variable: String country, HashBag bag, String done;

Body:

create filter ArrayList;

create RowFilter;

create an empty Scan instance;

pass filter ArrayList to scan;

get a scanner to iterate over the rows;

for (Result r: resultScanner) **{**

add filters to filter ArrayList;

close resultScanner;

for (Object o : bag) **{**

count number of products;

print number of product sold in given country;

}

}}

Output:

```
Processing Query 2...
Count Of Products In The Given Country :
In 'State-4' no of produts sold : 1613028
In 'State-1' no of produts sold : 3228613
In 'State-2' no of produts sold : 1612983
In 'State-3' no of produts sold : 1613985
Elapsed Time For Query 2 : 249110
```

Query 3. To find total no. of products sold in given store

Class Definition: public class ThirdQuery **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

call thirdQuery method; **}**

thirdQuery method: private static void thirdQuery **{**

Parameter passed: HTable client;

Local variable: String store, int count;

Body:

create filter ArrayList;

create ValueFilter;

create an empty Scan instance;

pass filter ArrayList to scan;

get a scanner to iterate over the rows;

for (Result r : resultScanner) **{**

Increment count;

print count of products sold in given store; **}**

close resultScanner;

}}

Output:

```
Processing Query 3...
Total no. of products sold in 'Store-330' are : 80656
Elapsed Time For Query 3 : 150600
```

Query 4. To find top three countries where given product has been sold.

Class Definition: public class FourthQuery **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

call fourthQuery method; **}**

fourthQuery method: private static void fourthQuery **{**

Parameter passed: HTable client;

Local variable: String productID, TreeBag bag, HashMap mm, InverseMap im;

Body:

create ArrayList;

create RowFilter;

create an empty Scan instance;

pass filter ArrayList to scan;

get a scanner to iterate over the rows;

for (Result r : resultScanner) **{**

add filters to ArrayList;

close resultScanner;

for (Object o : bag) **{**

call testMapInverse method;

}}}

Class Definition: public class InverseMap **{**

Body:

testMapInverse method: public void testMapInverse **{**;

Parameters passed: map, Boolean desc

Local variables: HashMap inverseMap, ArrayList topThreeList;

Body:

loop variables: Integer intValue, String key;

for (key : map.keySet()) **{**

intValue = map.get(key);

if (!map.containsKey(intValue)) **{** put intValue in inverseMap; **}**

add intValue to topThreeList; **}**

sort topThreeList;

if (desc) **{**

reverse topThreeList; **}}**

for (String key : keyList) **{**

print topThreeList;

}}}

Output:

```
Processing Query 4...
Top 3 countries where 'Product 3925' has been sold are :
Country-1 : 16077325
Country-6 : 8030406
Country-8 : 8038408
```

Query 5. To find total number of given category products sold in given store

Class Definition: public class FifthQuery **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

call fifthQuery method; **}**

fifthQuery method: private static void fifthQuery **{**

Parameter passed: HTable client;

Local variable: String store, String category, int count;

Body:

create filter ArrayList;

create ValueFilter;

create an empty Scan instance;

pass filter Arraylist to scan;

get a scanner to iterate over the rows;

for (Result r : resultScanner) **{**

increment count;

print count of products sold in given store; **}**

close resultScanner;

}}

Output:

```
Processing Query 5...
Total no. of Category-40 products sold in 'Store-981' are : 221
Elapsed Time For Query 5 : 146417
```

Query 6. To find count of particular product in the given country and the given state.

Class Definition: public class SixthQuery **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

call sixthQuery method; **}**

sixthQuery method: private static void sixthQuery **{**

Parameter passed: HTable client;

Local variable: String store, int count, HashBag bag, String done;

Body:

create filter ArrayList;

create ValueFilter;

create an empty Scan instance;

pass filterlist to scan;

get a scanner to iterate over the rows;

add filters to filter ArrayList;

for (Result r : resultScanner) **{**

add value in bag; **}**

close resultScanner;

for (Object o : bag) **{**

if (is done) **{**

print number of products sold in given country and given state; **}**

assign o to done; **}}}**

Output:

```
Processing Query 6...
Count Of Particular Product In The Given Country and The Given State:
In 'Store-283' no of produts sold : 3
In 'Store-282' no of produts sold : 1
In 'Store-285' no of produts sold : 1
In 'Store-284' no of produts sold : 1
In 'Store-490' no of produts sold : 1
In 'Store-288' no of produts sold : 2
In 'Store-188' no of produts sold : 1
In 'Store-187' no of produts sold : 2
In 'Store-189' no of produts sold : 2
In 'Store-390' no of produts sold : 1
In 'Store-184' no of produts sold : 1
In 'Store-183' no of produts sold : 1
In 'Store-186' no of produts sold : 1
In 'Store-488' no of produts sold : 2
In 'Store-385' no of produts sold : 2
In 'Store-193' no of produts sold : 1
In 'Store-388' no of produts sold : 2
In 'Store-381' no of produts sold : 1
In 'Store-599' no of produts sold : 1
In 'Store-382' no of produts sold : 2
In 'Store-383' no of produts sold : 2
In 'Store-595' no of produts sold : 1
In 'Store-596' no of produts sold : 1
In 'Store-483' no of produts sold : 1
In 'Store-389' no of produts sold : 1
In 'Store-487' no of produts sold : 1
```

Query 7. To do full table scan

Class Definition: public class FullTableScan **{**

Body:

Main method: public static void main(String[] args) **{**

create the required configuration;

instantiate new table reference;

create an empty Scan instance;

get a scanner to iterate over the rows;

print time taken to scan;

close resultScanner;

}}

Output:

Elapsed Time For Query 7: 323672

Chapter 5

Results and Analysis

Effect of using compression in HBase can be seen directly from the HBase master UI. Table 5.1 shows number of online regions in compressed tables is less than the uncompressed one. For snappy and LZO number of regions are same that means in both cases the size of compressed table is almost same. For GZIP number of the regions is the least which means it provides highest compression ratio among all three.

Table 5.1: Count of Regions

Table Name	No. of Regions
ret_LZO	5
ret_snappy	5
ret_gz	4
retail	7

Following command was used to check the length/size (bytes) of these tables and it was found that there is huge difference between the lengths of uncompressed table and compressed tables.

priyanka@priyanka:~/hadoop-1.0.4$ ./bin/hadoop dfs -dus /HBase/table

Table 5.2 gives the percentage of reduced data size (in bytes). Here again clearly GZIP is the winner as GZIP mainly aims at providing better compression ratios as stated earlier. Here too it was found performance of snappy and LZO is nearly equal. Figure 5.1 graphically represents the same thing but in comparison with the size of retail table. This analysis proves that GZIP can perform better in those database applications where storage requirements is the main concern.

Table 5.2: Space usage of Tables

Table Name	Data Size (Bytes)	Reduced Data Size (%)
ret_LZO	3244491342	69
ret_snappy	3262429218	68
ret_gz	2030768237	80

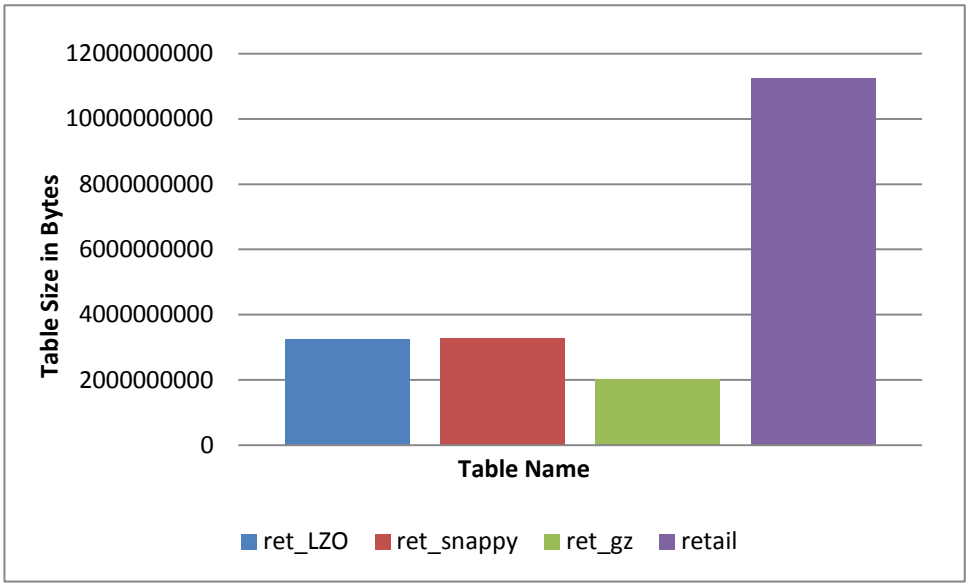


Figure 5.1: Data Size of Tables

Another important area where effect of compression can be seen is in the performance improvement of queries execution. Results shows that execution time required for executing queries, when using compression on table is less than the execution time needed for executing queries on uncompressed table.

Further, analysis of compression codecs used on HBase tables shows that snappy and LZO are much faster than GZIP.

Table 5.3 shows the queries execution speed up by using compression algorithms on tables in comparison with the uncompressed one.

Table 5.3: Execution Time Speed up for Queries

Query Number	Table Name	Execution Time Speed (%)
1	ret_lzo	41
1	ret_snappy	45
1	ret_gz	29
2	ret_lzo	4
2	ret_snappy	15
2	ret_gz	-0.4
3	ret_lzo	7.06
3	ret_snappy	21.6
3	ret_gz	10
4	ret_lzo	5.3
4	ret_snappy	3.7
4	ret_gz	-1.6

5	ret_lzo	18.9
5	ret_snappy	22
5	ret_gz	8.2
6	ret_lzo	18.2
6	ret_snappy	24.8
6	ret_gz	11.2
7	ret_lzo	11.9
7	ret_snappy	14.7
7	ret_gz	8.7

Figure 5.2 presents the comparison between executions speeds gain among the three codecs used in queries execution. Results are largely in favor of snappy. It shows snappy outperforms other two and is faster in executing of queries. It shows that in some queries GZIP execution time is even more than the execution time for uncompressed table retail. This shows that GZIP decompression rate is not speedy.

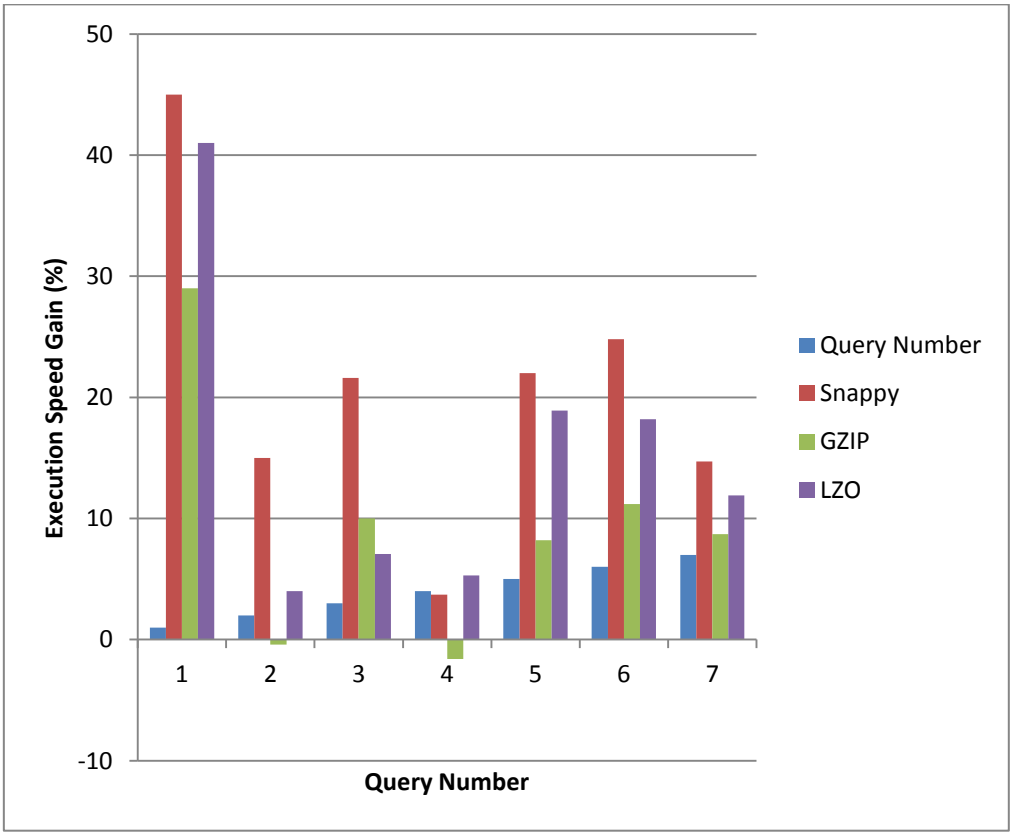


Figure 5.2: Comparison of Execution Speed of Queries using Compression

To analyze the effect of compression in terms of important resources like memory used, CPU utilization, network utilization and disk space used during the execution of queries on Hadoop and HBase, installation of Ganglia Monitoring System is required and. Working of Ganglia monitoring system can be briefly outlined as follow:

1. **Gmond: G**anglia **mon**itoring **d**aemon is installed on each node in the cluster (only one node in this work), that gathers the server and HBase metrics of that node.
2. **Gmetad:** Metrics gathered by Gmond are then subsequently polled to **G**anglia **met**a **d**aemon servers, where the metrics are computed and saved in round-robin database.
3. **PHP Web Frontend:** Installation of PHP web frontend on the same Gmetad server is done, so as to access Ganglia from web browser. Its PHP web frontend shows all the graphs of metrics of HBase that Ganglia has collected. It also shows metrics for cluster view, host view and node view.

Figure 5.3(a) presents the overview of HBase cluster. It gives detail of load percentage of cluster, no. of CPUs in cluster, no. of hosts up and down that are working and not working. It shows values of metrics captured in last hour, last day etc.

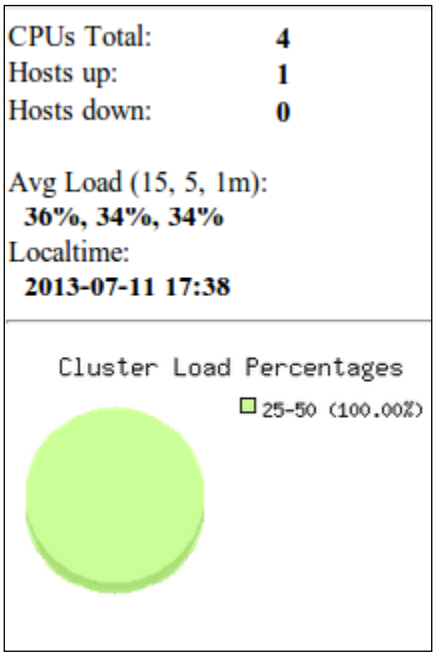


Figure 5.3(a): HBase Cluster Overview

Figure 5.3(b) shows memory used, during queries execution hour. It shows that all compression codecs does uses almost equal memory during query execution.

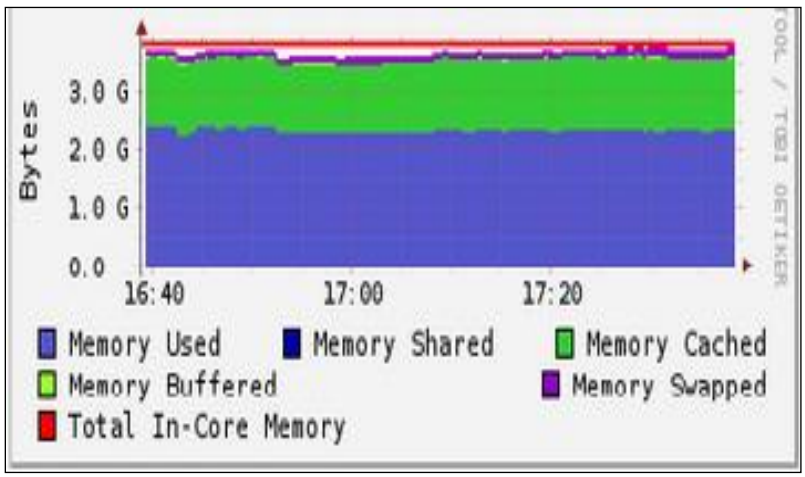


Figure 5.3(b): HBase Cluster Memory used during Query Execution

Figure 5.3(c) shows CPUs utilization on HBase cluster during queries execution hour. Graph shows that CPUs are utilized efficiently.

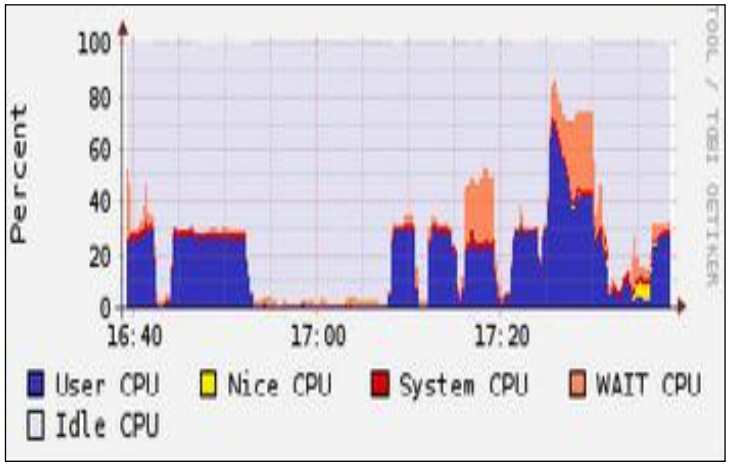


Figure 5.3(c): HBase Cluster CPUs utilization during Query Execution

Finally, Figure 5.3(d) shows network utilization on HBase cluster during queries execution hour.

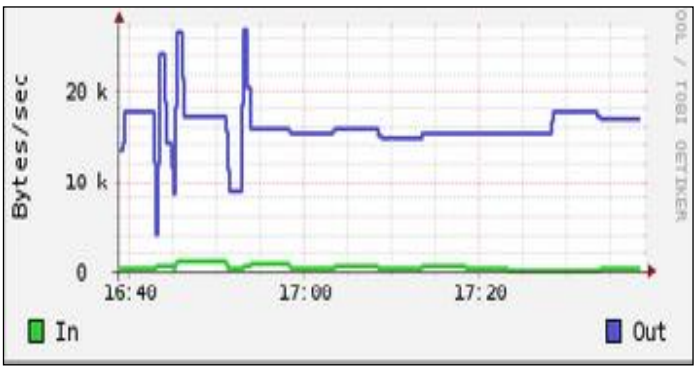


Figure 5.3(d): HBase Cluster Network utilization during Query Execution

CPU utilization: Efficient CPU resource usage is very important in applications which are I/O intensive and the compression ratio achieved is minimal. Figure 5.4 shows CPU user metric, it tells the usage of CPU at user level for each table (for query 5 and6). CPU usage in percentage is gathered for all queries executed on all tables.

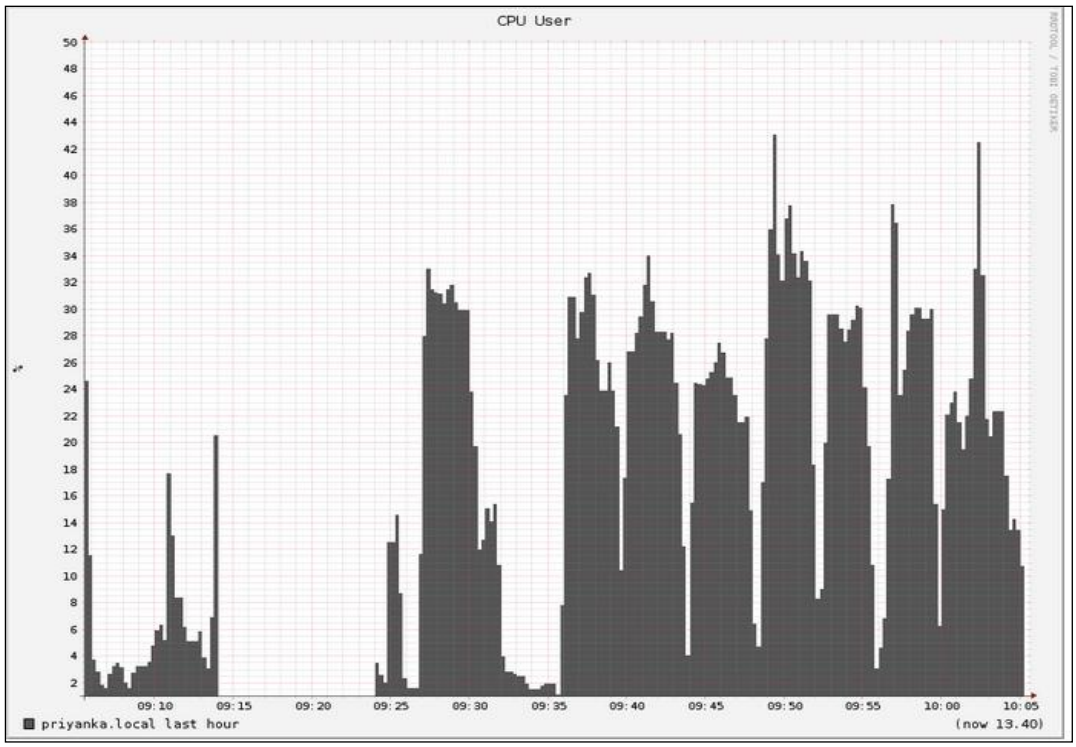


Figure 5.4: CPU usage for Query 5 and 6

Investment of CPU cycles in decompressing the data read from disk is required. Snappy and LZO invests less CPU cycles means that their decompression speed is faster than GZIP. Snappy and LZO are not CPU intensive whereas GZIP is.

Results have been tabulated under Table 5.4. Results shows that snappy have low CPU usage. LZO CPU usage percentage is comparable with snappy but results for GZIP shows maximum CPU usage.

Table 5.4: Comparison of CPU Utilization

Query Number	Table Name	CPU Utilization (%)
1	ret_LZO	29.5
1	ret_snappy	31
1	ret_gz	32
1	retail	33.5
2	ret_LZO	30.5
2	ret_snappy	26
2	ret_gz	31
2	retail	31.8
3	ret_LZO	31
3	ret_snappy	30
3	ret_gz	46.4
3	retail	26
4	ret_LZO	30
4	ret_snappy	26
4	ret_gz	32.6
4	retail	34
5	ret_LZO	37.9
5	ret_snappy	30
5	ret_gz	42.5
5	retail	42.1
6	ret_LZO	33
6	ret_snappy	32
6	ret_gz	34
6	retail	27.2
7	ret_LZO	32
7	ret_snappy	30
7	ret_gz	37
7	retail	32.5

Comparison of Snappy, LZO, GZIP in terms of CPU usage is shown in Figure 5.5.

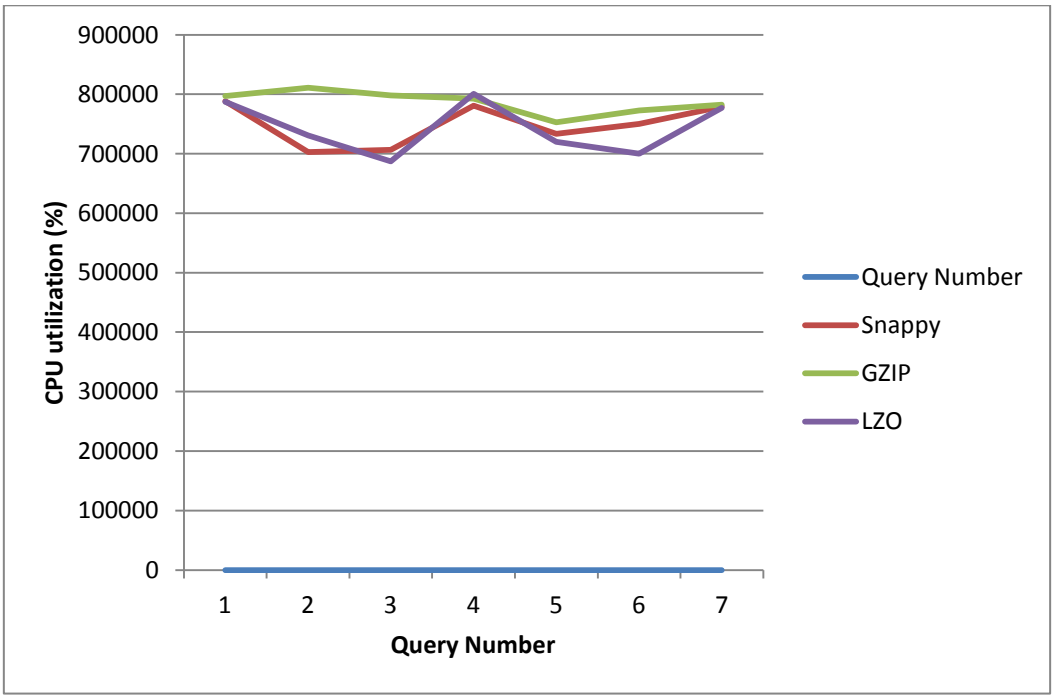


Figure 5.5: Comparison of CPU usage by Queries using Compression

From comparison graph it is clear that snappy saves CPU cycles yet provides fast compression and decompression speed.

Network utilization: Compressed data read from disk needs to be transferred over the network, compression ratio directly affects the number of roundtrips required to read compressed data over the wire. Compression on the sending end and decompression on the receiving end, in snappy and LZO is a pure win.

Figure 5.6 shows Network utilization metric; it tells the received and sent bytes per second (for query 5 and 6). Firstly all values have been collected for network usage in B/s for all queries executed on all tables. Results have been shown in Table 5.5.

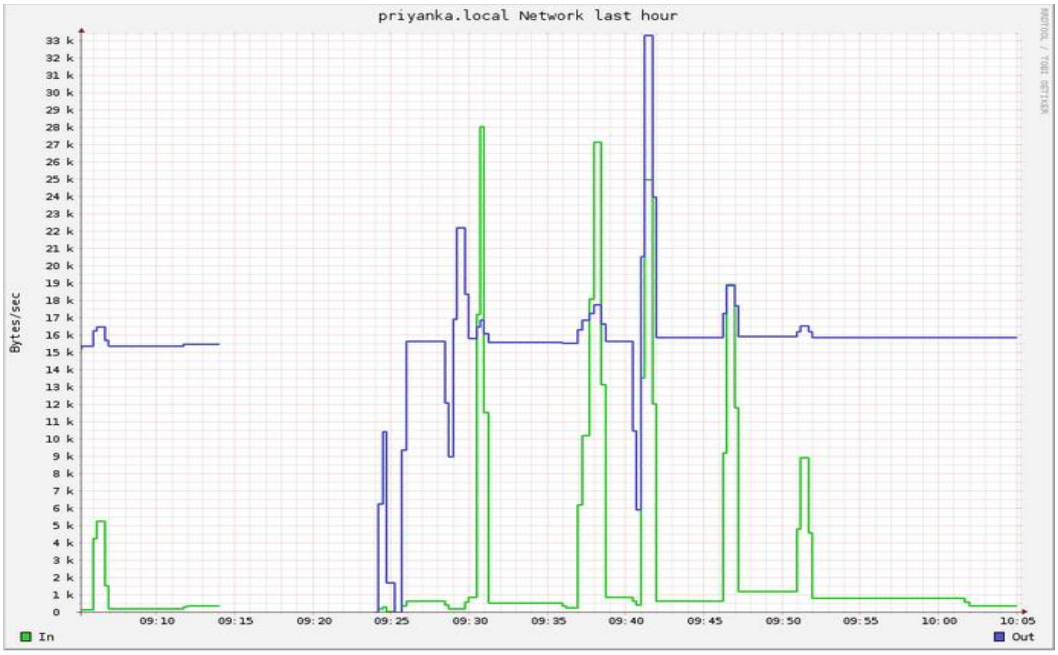


Figure 5.6: Network Utilization for Query 5 and 6

Figure 5.7 shows comparison of three codecs in terms of Network Utilization. Snappy and LZO uses the network bandwidth efficiently. Snappy and LZO are great for sending or receiving large pieces of data over the network.

Table 5.5 Comparison of Network Utilization

Query Number	Table Name	Network Utilization In/Out (Bytes/sec)
1	ret_lzo	9.5/108
1	ret_snappy	7.2/98
1	ret_gz	8/118
1	retail	10.2/125
2	ret_lzo	8/50
2	ret_snappy	8/48
2	ret_gz	8.5/94
2	retail	9.8/114
3	ret_lzo	0.5/15.7
3	ret_snappy	0.5/15.8
3	ret_gz	0.5/16.4
3	Retail	0.5/15.8
4	ret_lzo	3.5/12

4	ret_snappy	3.5/14
4	ret_gz	5/17
4	Retail	4.5/17
5	ret_lzo	1/16
5	ret_snappy	1/16
5	ret_gz	5/16.5
5	Retail	1/16
6	ret_lzo	0.5/15.5
6	ret_snappy	28/16.5
6	ret_gz	27/17.5
6	Retail	19/19
7	ret_lzo	11/17
7	ret_snappy	14/15
7	ret_gz	16/18
7	Retail	18/13

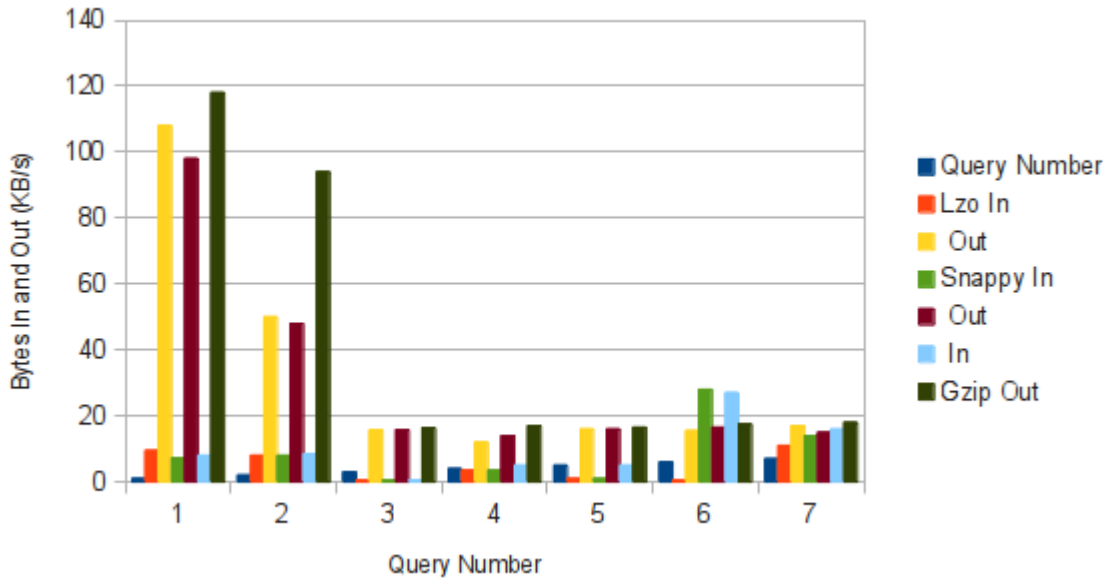


Figure 5.7: Comparison of Network usage by Queries using Compression

Memory usage: This section talks about memory used by queries during execution with compression enabled. How implementing compression helps in reducing memory usage would be of great help. Figure 5.8 shows Memory Used metric, captured from ganglia during queries execution; it shows the amount of cached memory in Bytes (for query 5 and 6).

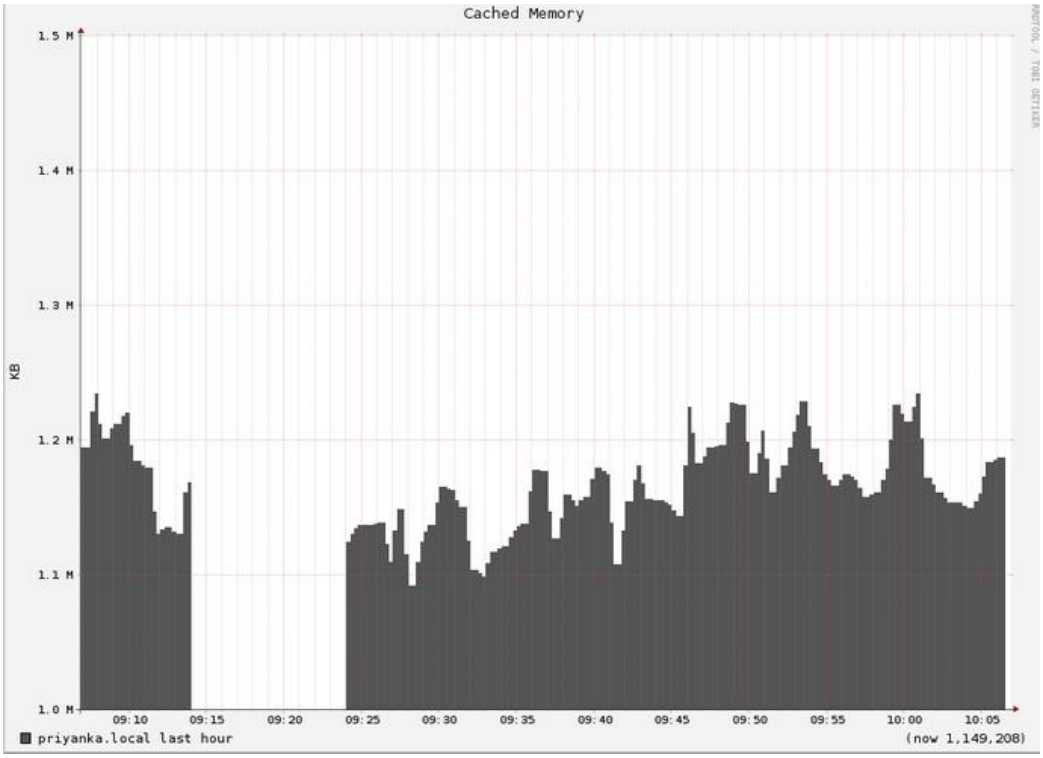


Figure 5.8: Amount of Memory Cached for Query 5 and 6

Results captured for all the queries corresponding to compression codecs are given in Table 5.6. In case of Snappy and LZO memory usage values came out less than in the case of GZIP. Here, LZO even outperforms snappy by slight amount. Figure 5.9 shows comparison of three codecs in terms of Memory usage.

Table 5.6: Comparison of Memory used

Query Number	Table Name	Memory Used (KB)
1	ret_lzo	786992
1	ret_snappy	788234
1	ret_gz	796932
1	retail	1314232
2	ret_lzo	730544
2	ret_snappy	702564
2	ret_gz	810980
2	retail	1321642
3	ret_lzo	686895
3	ret_snappy	706532
3	ret_gz	798037
3	retail	1149232
4	ret_lzo	800514
4	ret_snappy	780820
4	ret_gz	792562
4	retail	1423520
5	ret_lzo	720114
5	ret_snappy	733572
5	ret_gz	752860
5	retail	1323602
6	ret_lzo	700128
6	ret_snappy	750211
6	ret_gz	772910
6	retail	1223698
7	ret_lzo	776903
7	ret_snappy	779032
7	ret_gz	782561
7	retail	15720163

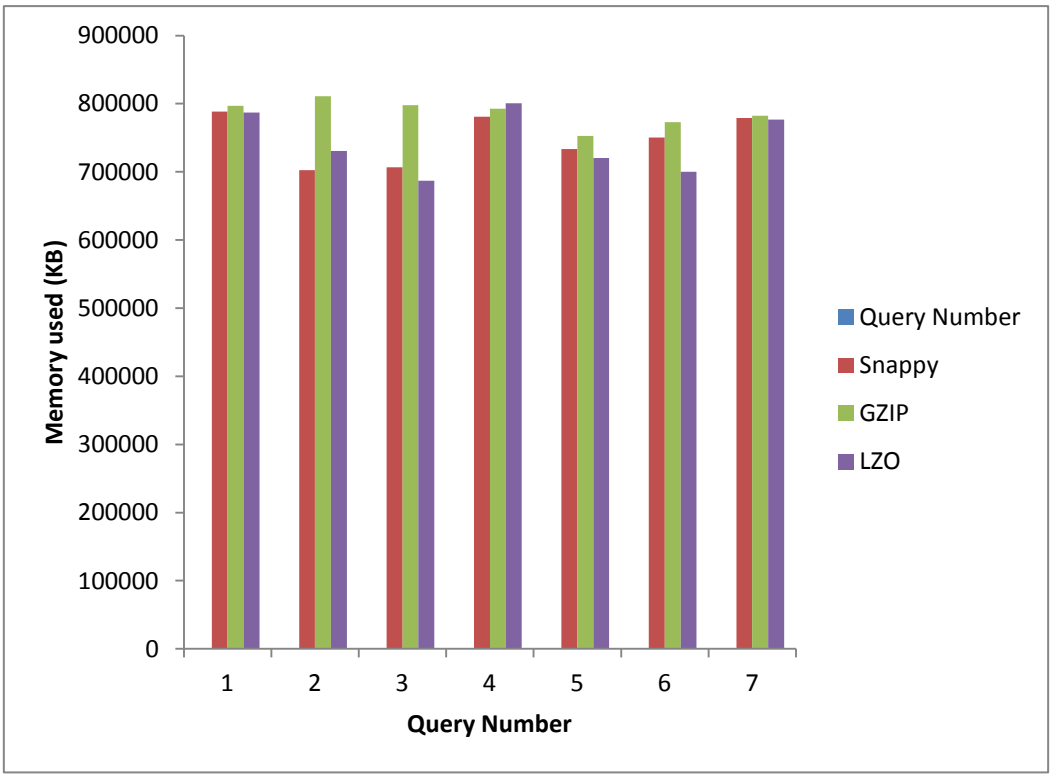


Figure 5.9: Comparison of Memory used by Queries using Compression

Fast compression codecs save memory, so snappy and LZO outperforms GZIP.

Next chapter summarizes this thesis work and conclusions drawn along with future work.

Chapter 6

Conclusion and Future Scope

Conclusion

For large clusters and large jobs, compression can lead to substantial benefits. Compression helps in improving overall performance of the NoSQL column oriented data-stores by optimally saving memory used and network bandwidth. Following conclusions are drawn from this work can be summarized as follows:

1. Compression in HBase improves query execution speed. Among the three taken codecs, Snappy performs best in reducing time taken by queries to execute.
2. For human readable text, Snappy and LZO are faster in compress and decompress time but less efficient in terms of compression ratio. Whereas GZIP gives higher compression ratios (7% higher than snappy) but is not so fast.
3. Snappy and LZO can perform well in applications that are I/O intensive whereas GZIP can be used in the application that starves on disk capacity.
4. Compression in HBase cause low usage of RAM/ Memory for no additional cost.
5. Since Snappy and LZO have fast decompression speed, thus not taking too many CPU cycles. Snappy and LZO have low CPU usage. Whereas GZIP has high value of CPU utilization.
6. Compressed data read from disk needs to be transferred over the network, compression ratio directly affects the number of roundtrips required to read compressed data over the wire. Snappy and LZO efficiently use network bandwidth since they are fast than GZIP.

Future Scope

In future this work can be extended as follows:

1. As in this work effect of compression has been shown by using human readable text data; in future it can be extended to other types of data like images.
2. A deep research can be done on read and write patterns of HBase and design of a new custom compression codec from the scratch to get better performance than the existing codecs.
3. Query executer can be built for HBase which can directly execute queries on compressed data, thus saving I/O requirements spend in decompression of data before executing queries.

References

[1] Lars George, *HBase: The Definitive Guide*. CA: O'Reilly Media, 2011.

[2] Orenstein, Gary. "What the Heck Are You Actually Using NoSQL for?" *High Scalability.com* Web. Dec. 6, 2010. *http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html*,

[3] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira, "Integrating Compression and Execution in Column-Oriented Database systems," *in Proc. of* ACM *SIGMOD International Conf. on Management of Data*, pp. 671-682, June 27-29, 2006.

[4] Nick Dimiduk and Amandeep Khurana, *HBase in Action*. New York: Manning Publications Co., 2012.

[5] Chris Eaton, Dirk Deroos, Tom Deutsch, George Lapis and Paul Zikopoulos, "*Understanding Big Data*: *Analytics for Enterprise Class Hadoop and Streaming Data*". New York : Mc Graw Hill, 2011.

[6] "CAP Theorem," *Wikipedia, the Free Encyclopedia.* Web. July, *2013.* *http://wikipedia.org/wiki/CAP_theorem.html*

[7] "NoSQL," Web. July, 2013. *http://nosql-database.org.html*

[8] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data," *in Proc of 7th Symposium on Operating System Design and Implementation*. Seattle, WA. Google, Inc., Nov. 2006. Web. 25 Jan. 2011.

[9] "HBase," Web. July, 2013 *http://hbase.apache.org/html*

[10] "Hypertable," Web. July, 2013 *http://hypertable.org/html*

[11] "Cassandra," Web. July, 2013 *http://cassandra.apache.org/html*

[12] "Introduction to Hbase," Web. July, 2013 *http://hadoop-hbase.blogspot.in/2011/12/introduction-to-hbase.html*

[13] Shashank Tiwari, *Professional NoSQL*. Canada: John Wiley & Sons, Inc., 2011.

[14] "Apache Zookeeper." *http://zookeeper.apache.org.html*

[15] Tom White, *Hadoop: The Definitive Guide*, 3rd ed. CA: O'Reilly Media, Inc., 2012.

[16] Zach Pratt, "A Review Of Column Oriented Data stores."

[17] D.J. Abadi, "Query execution in column-oriented database systems," PhD dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering & Computer Science, 2008.

[18] "LZO." Web July , 2013. http://www.oberhumer.com/opensource/lzo/html

[19] Yifeng Jiang, *HBase Cookbook Administration*. Birmingham, UK: PACKT Publishing, 2012.

[20] "Snappy." Web. July, 2013. http://code.google.com/p/hadoop-snappy/html

[21] "Gzip." *Wikipedia, the free encyclopedia*, Web. July, 2013. http://en.wikipedia.org/wiki/Gzip/html

[22] G.Graefe and L.D Shapiro, "Data compression and database performance," *in Proc. of ACM/IEEE- Computer Science Symposium on Applied computing* pp. 22 -27, April 1991.

[23] B. R. Iyer and D. Wilhite, "Data compression support in databases," *in Proc. of 20th International Conf. on Very Large Data Bases* (VLDB), pp. 695–704, Sept. 12-15, 1994.

[24] J. Goldstein, R. Ramakrishnan and U. Shaft, "Compressing relations and indexes," *in Proc. of 14th International Conf. on Data Engineering* (ICDE), pp. 370–379, Feb. 23-27, 1998.

[25] Theodore Johnson, "Performance measurements of compressed bitmap indices," *in Proc. of 25th International Conf. on Very Large Data Bases* (VLDB), pp. 278–289, Sept. 7-10, 1999.

[26] Gautam Ray, Jayant R. Haritsa, and S. Seshadri, "Database compression: A performance enhancement tool," *in Proc. of 7th International Conf. on Management of Data (COMAD)*, Dec. 28-30, 1995.

[27] C. A. Lynch and E. B. Brownrigg, "Application of data compression to a large bibliographic data base," *in Proc. of 7th International Conf. on Very Large Data Bases* (VLDB), pp. 435–447, Sept. 9-11, 1981.

[28] Patrick E.O'Neil and Dallan Quass, "Improved query performance with variant indexes," *in Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 38–49, May 13-15, 1997.

[29] Peter A. Boncz, Stefan Manegold and Martin L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," *in Proc. of 25th International Conf. on Very Large Data Bases (VLDB*), pp. 54–65, Sept. 7-10, 1999.

[30] Keshang Wu, Ekow Otoo, and Arie Shoshani, "Compressed bitmap indices for efficient query processing" Technical Report LBNL-47807, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[31] Keshang Wu, Ekow Otoo, Arie Shoshani, and H. Nordberg, "Notes on design and implementation of compressed bit vectors," Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[32] Zandi, Balakrishna R. Iyer, and Glen G. Langdon Jr., "Sort order preserving data Compression for Extended Alphabets," *in Proc. of IEEE International Conf. on Data Compression*, pp. 330–339, Mar.30 - Apr.1, 1993.

[33] Alistair Moffat and Justin Zobel, "Compression and fast indexing for multi-gigabyte text databases," *Australian Computer Journal*, Vol. 26, no. 1, pp.1–9, 1994.

[34] Roger MacNicol and Blaine French, "Sybase IQ multiplex - designed for analytics," *in Proc of 30th International Conf. on Very Large Data Bases*(VLDB), pp. 1227– 1230, Aug.31- Sept.3, 2004.

[35] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley. B. Zdonik, "C-Store: A column-oriented DBMS," *in Proc. of 31st International Conference of Very Large Data Bases (VLDB)*, pp. 553–564, Aug.30 – Sept. 2, 2005.

[36] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sandor Heman, "MonetDB/X100 - A DBMS in the CPU Cache," *IEEE Data Engineering Bulletin*, Vol. 28, pp.17–22, June 2005.

[37] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz, "Super-scalar RAM- CPU cache compression," *in Proc. of 22nd International Conf. on Data Engineering*, ICDE, pp. 59, Apr. 3-8, 2006.

[38] Jacob Ziv and Abraham Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, Vol. 23, no.3, pp. 337–343, May 1977.

[39] Jacob Ziv and Abraham Lempel, “Compression of individual sequences via variable- rate coding,” *IEEE Transactions on Information Theory*, Vol. 24, no.5, pp. 530–536, Nov. 1978.

List of Publications

1. Priyanka Raichand and Rinkle Rani, "Query Execution and Effect of Compression on NoSQL Column Oriented Data-store Using Hadoop and HBase," *International Journal of Scientific and Engineering Research (IJSER)*,ISSN 2229-5518 **[Accepted]**.