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

Priyanka Raichand, Rinkle Rani

Abstract— 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 paper is to see and analysize 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.

Index Terms— Column oriented data-store, Compression, Gzip, Hadoop, HBase, NoSQL, LZO, Snappy.

1 INTRODUCTION

Today everyone is connected over the Internet and look to find relevant results instantaneously. 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].

- *Priyanka Raichand is currently pursuing masters degree program in computer science and engineering in Thapar University, India, E-mail: raichand.priyanka@gmail.com*
- *Rinkle Rani is assistant professor in Thapar University,India, E-mail: ragarwal@thapar.edu*

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.

In this paper, effect and analysis of compression techniques is done using query execution on NoSQL column oriented datastore HBase. In next section, data model of HBase is

briefly explained. Section 3 answers the basic questions that why compression is needed and what are the various data compressing codecs well suited for HBase. Section 4, contains the implementation part. Section 5 layouts the analysis that has been taken out on the basis of query execution. Conclusions and future scope is offered in Section 6.

2 HBase- Data Model

HBase [3] is a distributed, persistent, strictly consistent, sparse, open source storage system. It is multidimensional-sorted map that 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. The key sorts these cells. It is quite important property as it allows for searching rather than just retrieving a value for a known key. It is multidimensional means that the key itself has structure. Each Key has following 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. One attractive feature of HBase is that it is distributed. The data can be spread over 100s or 1000s of machines and HBase manages the load balancing automatically. HBase do load shifting gracefully and transparently to the clients. We can define in brief the basic constructs of HBase data model as follows and fig1. displays the same:

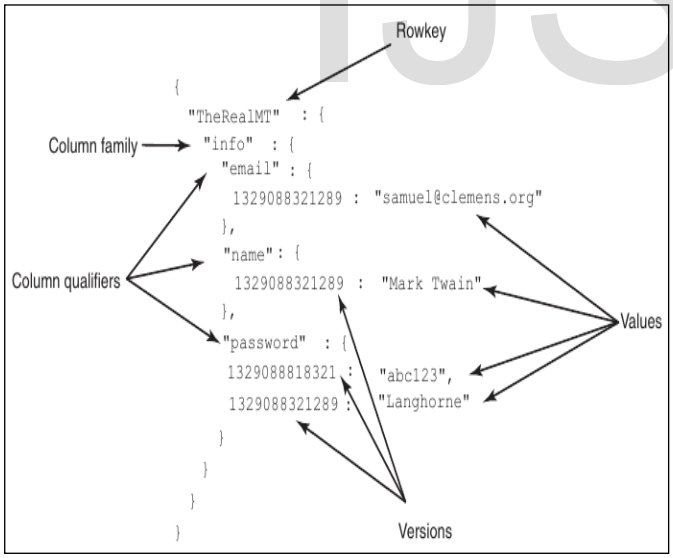

Fig1: HBase Logical Data Model

A. Table: Applications data is stored into an HBase table. Tables are made of rows and columns. The intersection of row and column coordinates known, as cells are versioned.

B. 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.

C. Row Key: The rowkey is 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.

D. 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.

HBase provides Bigtable [4] like capabilities on top of Hadoop and HDFS. HBase uses many filesystem like local (for stand alone mode), S3, HDFS and other. But mainly for HBase HDFS is the best filesystem, as it has all the required features since HDFS takes full advantage of Map Reduce parallel, streaming access support. HBase is capable of storing structured and semistructured data. HBase make use of existing system, like HDFS and ZooKeeper, but also adds its own layers to form a complete platform.

3 Why Compression and Compression Codecs

This section concisely describes the need of compression in HBase and then the various compression codecs available. File compression results in two major benefits: it reduces the space needed to store files, and it also speeds up data transfer rate across the network, or to or from disk. Both of these savings can be significant, when dealing with large volumes of data. Using some form of compression for storing data lead to an increase in IO performance. Hadoop workloads are generally data-intensive, so making the data reads a bottleneck in overall application performance. By using compression we reduce the size of our data achieving faster reads. It is simply trading I/O load for CPU load as we need to uncompress that data too so we use some 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.

3.2 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 helps saving disk I/O and instead paying with a 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 [5]. 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 as follows:

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

3.2 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.

LZO: Lempel-Ziv-Oberhumer (LZO) [6] 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 [7].

Table1: Comparison of Compression Algorithms by Google

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

Snappy: Snappy [8] 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.

GZIP: The GZIP [9] 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.

4 IMPLEMETATION

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. 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. 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.

4.1 Loading Bulk Data in HBase Table

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. Fig2. shows schema of the all tables.

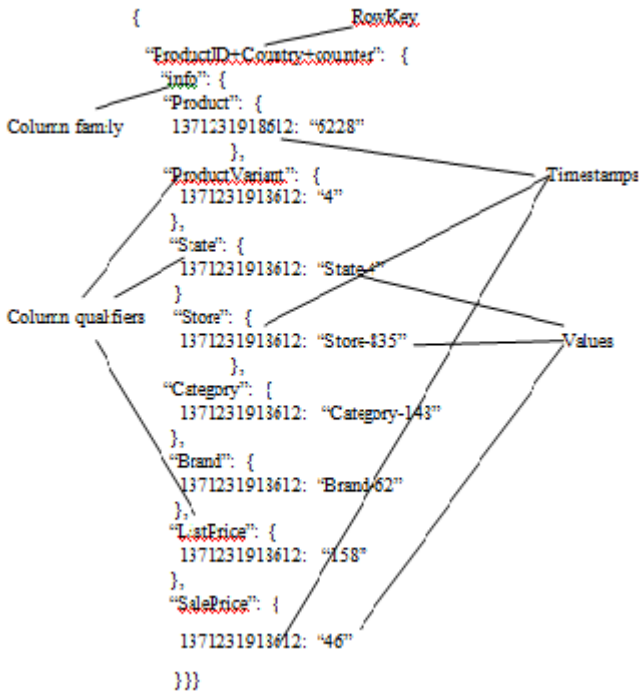

Fig 2: HBase Table Schema

4.2 Map Reduce Job for Bulk Data Loading

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. These computations are divided into two categories: the map and the reduce computations. 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 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
}}

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

- Installation of Apache Ant, Maven, libtoolize.
- Get the latest hadoop-lzo/ hadoop-snappy.
- Build hadoop-lzo / hadoop-snappy from source.
- Build the native and Java hadoop-lzo/ hadoop-snappy libraries from source, depending on OS.
- This step creates the hadoop-lzo/ hadoop-snappy build/native directory and the hadoop-lzo / build/ hadoop-lzo-1.0.4.jar file.
- 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.
- 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. In next step 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 Query Execution

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 two queries followed by output snapshots. Only two queries are shown here.

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. 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

Table Names	Reduced Data Size (%)
ret_lzo	69
ret_snappy	68
ret_gz	80

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
```

5 Results and Analysis

Effect of using compression in HBase can be seen directly from the HBase master UI. Table 2 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 2: 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 3 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.

Table 3: Space usage of Tables

Fig3. 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.

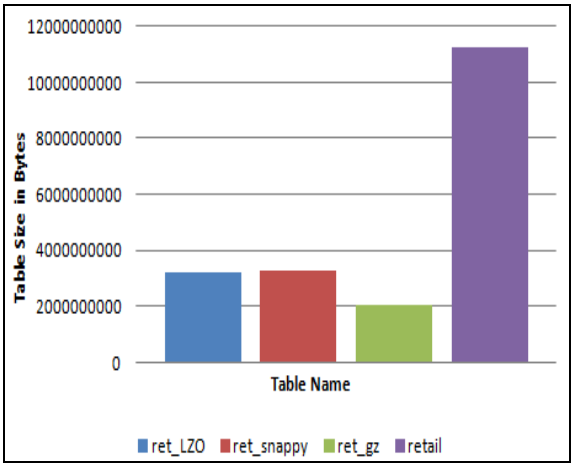

Fig 3: Comparison of Table Size

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. For this performance evaluation 7-8 queries were executed, here only results of two queries are shown.

Table 4 also shows the queries execution speed up by using compression algorithms on tables in comparison with the uncompressed one only for prior mentioned two queries.

Table 4: Execution time Speed up

Query Number	Table Name	~Execution Time Speed Up (%)
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

Further, analysis of compression codecs used on HBase tables shows that snappy and LZO are much faster than GZIP. Fig4. 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.

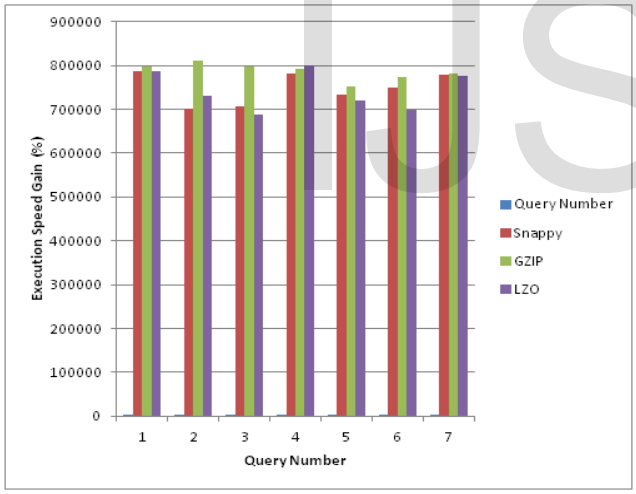

Fig 4: Comparsion of Query Execution Speed using Compression

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. Table 5 shows the values of all metrics captured for query 1 and 2 against each table enabled with compression.

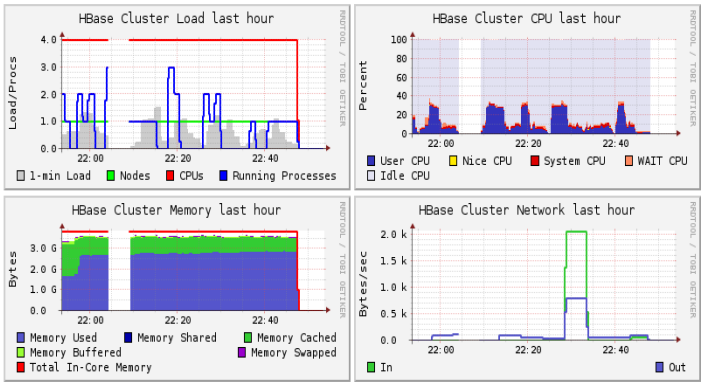

Fig 5: HBase Cluster Overview during Queries Execution

CPU Utilization: Efficient CPU resource usage is very important in applications which are I/O intensive and the compression ratio achieved is minimal. CPU usage in percentage is gathered for all queries executed on all tables.

Table 5. Comparsion of Memory, Network and CPU utilization by Compressed Tables during Query Execution (1&2)

Query Number	Table Name	Memory Used (KB)	Network Utilization In/out (B/sec)	CPU Utilization (%)
1	ret_lzo	786992	9.5/108	29.5
1	ret_snappy	760234	7.2/98	31
1	ret_gz	796932	8/118	32
2	ret_lzo	730544	8/50	30.5
2	ret_snappy	702564	8/48	26
2	ret_gz	810980	8.5/94	31

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. Fig6. depicts that Snappy and LZO are not CPU intensive whereas GZIP is.

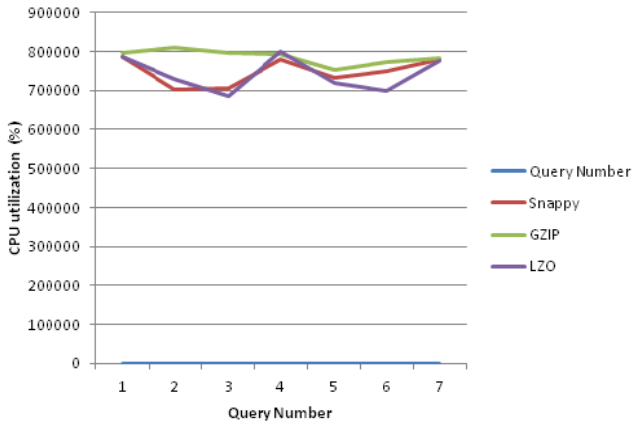

Fig 6: Comparison of CPU utilization

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. Snappy and LZO uses the network bandwidth efficiently. Fig7. shows that Snappy and LZO are great for sending or receiving large pieces of data over the network.

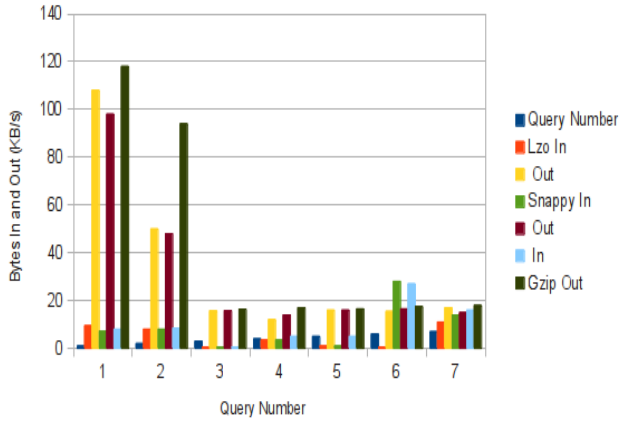

Fig7. Comparison Result of Network used

Memory Usage: How implementing compression helps in reducing memory usage would be of great help. Fig8. shows that Snappy and LZO memory usage values came out less than in the case of GZIP. Here, LZO even outperforms snappy by slight amount.

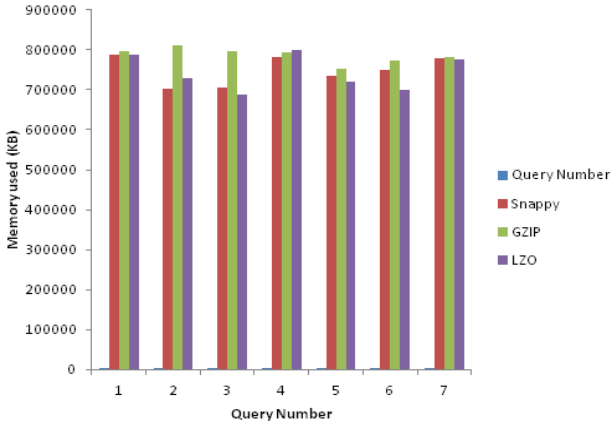

Fig 8: Comaprison of Memory Utilization

Next section presents the summary and conclusion of this paper along with future scope.

6 Conclusion and Future Scope

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:

Compression in HBase improves query execution speed. Among the three taken codecs, Snappy performs best in reducing time taken by queries to execute. For human readable text, Snappy and LZO are faster in compress and decompress time but less efficient in terms of compression ratio. GZIP gives higher compression ratios (7% higher than snappy) but is not so fast. 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. Compression in HBase cause low usage of RAM/ Memory for no additional cost. 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. 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.

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

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

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

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

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

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