Impact of Data Compression on the Performance of Column-oriented Data Stores

Tsvetelina Mladenova[1], Yordan Kalmukov[2], Milko Marinov[3], Irena Valova[4]
Department of Computer Systems and Technologies
University of Ruse, Ruse, 7017, Bulgaria

***Abstract*—Compression of data in traditional relational database management systems significantly improves the system performance by decreasing the size of the data that results in less data transfer time within the communication environment and higher efficiency in I/O operations. The column-oriented database management systems should perform even better since each attribute is stored in a separate column, so that its sequential values are stored and accessed sequentially on the disk. That further increases the compression efficiency as the entire column is compressed/decompressed at once. The aim of this research is to determine if data compression could improve the performance of HBase, running on a small-sized Hadoop cluster, consisted of one name node and nine data nodes. Test scenario includes performing Insert and Select queries on multiple records with and without data compression. Four data compression algorithms are tested since they are natively supported by HBase - SNAPPY, LZO, LZ4 and GZ. Results show that data compression in HBase highly improves system performance in terms of storage saving. It shrinks data 5 to 10 times (depending on the algorithm) without any noticeable additional CPU load. That allows smaller but significantly faster SSD disks to be used as cluster's primary data storage. Furthermore, the substantial decrease in the network traffic is an additional benefit with major impact on big data processing.**

Keywords*—Column-oriented data stores; data compression; distributed non-relational databases; benchmarking column-oriented databases

I. Introduction

The main factors that put pressure on the centralized relational model and play a key role in the development of NoSQL DBMS are: Volume, Velocity, Variety and Agility [1]. Volume and Velocity are referred to the possibility of management of big sets of data that is generated with high speed and should be processed fast. Variety is the existence of various structures and formats of data (structured, semi-structured and unstructured). Agility is the likelihood of an organization reacting to changes in the business [2].

Usually, the relational database management systems are seen as the best possibility of data storage, data retrieval and data processing. Regardless, the constant growing needs of scalability and the emerging needs for specific applications posed various challenges to the conventional relational databases [3,4]. For example, the continuous growth of user-generated data led to an increase in the types (structured, semi-structured and unstructured) and the volume of the data that has to be stored and processed. In practice, the fast growth in the volume of the data created a big problem for the conventional databases that require a pre-known defined schema of the data, based on relations. In fact, the pre-known schema became inadequate in many different scenarios.

The desire to meet this challenge led to the creation of NoSQL DBMS [2,5]. For the purposes of the distributed databases, NoSQL is defined in [6] as follows: NoSQL is a set of ideas and concepts that allow for fast and efficient processing of datasets with an emphasis on performance, reliability and speed. One of the challenges for the users of NoSQL is that there are a large number of different architectural data models to choose from (key-value, column-oriented, document-based, graph stores).

Compression of data elements in traditional database management systems significantly improves system performance by [7,8]: (1) decreasing the size of the data, (2) increase the productivity of input/output operations by reducing search time, (3) reduction of data transfer time in the communication environment (less data is transferred to the computer network) and (4) increasing the relative speed of the input-output buffer. For requests that are limited in terms of data input or retrieval speed, the CPU load resulting from the decompression is offset by I/O system improvements at the operating system level. All this has a significant effect when working with large amounts of data.

In the column store, each attribute is stored in a separate column, so that the sequential values of this attribute are stored sequentially on disk [4,9]. This data model provides more opportunities for improved performance as a result of applying algorithms to compress data elements compared to row-oriented architectures. In the column-oriented model, the use of different compression schemes that encode multiple values at once is natural. In tuple-oriented systems, such schemes do not work well because the individual attribute is stored as part of the whole tuple, so combining the same attribute from different tuples together into one value will require some way to "mix" tuples.

The main objective of the current research is to test the capabilities of a small-sized cluster when performing the operations Insert and Select to Hbase. The operations are performed with and without the supported Hbase compression algorithms. This article aims to observe the behavior of the cluster when performing the experiments.

The article is structured as follows: Section 2 reviews some related research. Section 3 presents column-oriented data model properties. Section 4 outlines the nature of Hbase.

Section 5 presents the conducted experiment. Finally, in Section 6 the authors make conclusions and offer ideas for further research.

II. Related Work

Many authors are doing extensive research on the topic of data compression in Hbase and are proposing some additional algorithms and strategies for optimization of the compression process.

To avoid the hot-spot issue in the Regional Servers, the authors in [10] are proposing sorting of the data before the compression. Another part of the proposed solution is the usage of different compression algorithms for different data types.

In [11] the authors are overviewing some common compression algorithms and through experiments are comparing the methods. The results from these experiments are used for the decision of how to compress a particular data column. As a result, they propose an architecture for query executor. The authorsalso pay attention to the order of the columns.

A short survey of data compression techniques for column-oriented databases is presented in [12]. The results show that the LZO and Snappy compressions have comparable compression rates and speeds.

An analysis on compression of data, presented in relational format is done in [13] resulting in the authors proposing a strategy for compressing similar data together. The authors are noting that the traversal strategy has a significant impact on the compression ratio.

The compression algorithms of Hbase are considered in [14]. A series of experiments are conducted and several aspects are analyzed - used memory, CPU utilization, network utilization and disk space usage. The results show that the usage of some form of compression is beneficial when dealing with large clusters and large jobs. In terms of query time execution, Snappy is the best performing algorithm.

III. Column-oriented Data Model Properties

The systems based on column families are important architecture models of NoSQL data because they can easily expand (scale horizontally) while processing big volumes of data. Column-oriented data models can be viewed as one multi-dimensional table (Table I), to which queries can be made using the primary key. In this type of data storage, the key can be the identifier of the row (Row ID) and the name of the column. The first dimension of the table is the RowKey, which denotes the row of the physical storage of the data in the table. The second dimension is the column family, which is similar of the attributes in the relational data model. The third dimension is the qualificator of the column. In theory, every column can have an unlimited number of qualificators. The fourth dimension is the timestamp which is automatically assigned and represents the moment of the cell creation in nanoseconds. The structure of the key can be seen in the following manner:

The advantages of column-oriented systems are as follows [1,2,5,9]:

- A high degree of scalability – in essence, column-oriented systems have the property of scalability, which means that when there is a high intensity of data addition, more general-purpose nodes have to be added to the cluster. With a good design of the cluster can be achieved a linear relation between the way the data is expanding and the number of the needed nodes. The main reason for this relation is the manner in which the Row identifier and the column names are being used when identifying a cell. With the maintenance of a simple interface, the system which works in a background mode, can distribute the queries between a large number of nodes without the need of performing any merge and join operations.
- A high degree of availability – by building a system that is easily scalable, the need for data duplication arises. Because column-oriented systems use effective communication, the replication cost is low. The lack of join operations allows the storage of individual portions of the column matrix on remote servers. This means that if one of the nodes that contain some of the matrix data, falls, and the other nodes will provide the services to these cells.
- Easy data addition – the column-oriented model does not require a pre-detailed design before the data insertion begins. The column names should be known prior to the insertion, but the identifier of the rows and the column qualifiers can be created at any given moment.

Improved data compression – storing data from the same domain increases the locality of the processing and thus improves the degree of data compression (especially if the attribute is sorted). The requirements for the transfer speed in the computer network are further reduced when transferring compressed data. The coefficient of the compression is higher in the column-oriented data storages as the sequential values that are being stored in the column are from the same data type, while the neighboring attributes in the tuple can be different types. When every value in the column has the same size, the speed of decompression can be high, especially when taking into account the super-scalar properties of today's processors. The columns can be further sorted which in turn will increase the potential of compression. The compression methods used in the column-oriented systems improve the CPU performance by allowing data manipulators to work directly on the compressed data without having to decompress.

TABLE I. Key Structure in Column-oriented Systems

Key				
Row-ID	Column family	Column qualifier	Time stamp	**Value**

IV. HBase Overview

Apache HBase (http://hbase.apache.org) is a distributed, fault-tolerant, column-oriented database modeled after Google's Bigtable [15]. Similarly, as Bigtable is built on top of the Google's distributed file system, HBase relies on the data storage provided by the Hadoop's distributed file system (HDFS). It is an open-source software, written in Java. Its primary goal is to provide random, fast (real-time) read/write access to very large tables (containing billions of rows and millions of columns) running on clusters of cheap commodity servers.

As a non-relational database, it does not rely on SQL to access the data; however the Apache Phoenix project [16] offers an additional SQL layer on top of HBase, and a JDBC driver that allows external JAVA applications to communicate with the database. Furthermore, data are accessible through Java API, REST API and Avro or Thrift gateway APIs.

HBase features linear horizontal scalability, strictly consistent reads and writes, automatic failover support between RegionServers, data compression, in-memory operation, and Bloom filters on a "per-column" basis as implemented in Bigtable [10].

HBase is a widely deployed database, used by large number of companies, ranging from IT giants (Facebook, Twitter, Yahoo!, Adobe, Meetup, OCLC – WorldCat's data storage, and others) to smaller companies and research projects. Before 2018 Facebook fully relied on HBase to store private messages (multimedia content) exchanged through "Messenger". Then they switch to MyRocks - Facebook's open source database project that integrates RocksDB as a MySQL storage engine [17]. Twitter uses HBase as a distributed backup of their production database (implemented in MySQL). Benipal Technologies utilizes a 35-node HBase cluster that holds over 10 billion rows with hundreds of datapoints per row. They compute over 1018 calculations daily using MapReduce directly on HBase.

Data compression is a method that is used to improve the performance and reduce the storage space. Hbase supports four different compression algorithms, which can be directly applied on the ColumnFamily. That includes SNAPPY [18], LZO [19], LZ4 [20] and GZ [21] compressions. When creating a table every ColumnFamily is defined separately meaning that some families can have a compression algorithm applied to them and some may not. Using a compression algorithm can lead to a significantly reduced storage space as it is seen from the conducted experiments.

V. Experimental Analysis

A. Experimental Environment and Dataset

A cluster of 10 servers is being used to conduct this experiment. A Name Node having two CPUs: Intel Xeon Silver 4110, 8 cores and a total of 32 threads, and 64GB RAM; 9 data nodes – each data node have one CPU Intel Xeon E-2124, 4 cores and 4 threads, and 16GB RAM; and 1 additional server, having the same technical specifications as the data nodes, that is not being featured in the cluster but is located in the same network as the cluster.

Every experiment consists of a Python code that initiates an Insert or Select queries to the Name Node. The python code is developed with the help of some additional Python libraries and packages. Most important of which is the HappyBase library. HappyBase is a library, developed on the ThriftPy2 Package. Essentially, the ThriftPy2 is used to establish the connection between the cluster (more precisely - the Name Node) and the client. All of the test scenarios are being performed from the additional server, the one that is located in the same network but is not featured in the cluster.

The data that is being sent and retrieved to the server consist of over 4 900 000 different sensor measurements. The description of the dataset can be seen in Table II.

TABLE II. Dataset Description

Attribute	**Column Family**
house_id	house:mac
date_time	house:datetime
ch1_watts to ch3_watts	ch:1 to ch:3
temp	readings:temp
gas_reading	readings:gas
appliance_1 to appliance_10	app:1 to app:10

While performing the experiment the sequential data transmission from the sensors is stimulated, as can be expected in real-life scenarios. Therefore, all of the Insert tests are done in the manner of several thousand queries being sent to the server instead of performing a single Load Dump. We strive to test the capabilities of the cluster as close to real-life situations as possible. Nevertheless, the experiments are performed in an almost sterile environment, as the cluster does not handle any additional tasks while inserting and selecting the data.

B. Comparative Experiment on Query Execution Time

1) Insert queries: The Insert Test Cases are done under the following conditions:

- The table that will store the row is created at the beginning of the test.
- The number of the versions of the cells is left at the default value – three.
- A total of fifteen Insert tests cases are performed – three tests without any compression and twelve tests with compression (four compressions multiplied by three times).
- Every test is performed with 200 000, 400 000, 600 000, 800 000 and 1 000 000 different queries send to the server.

The inconsistent manner of the insert query execution is notably visible in these tests. The number of requests per second is around 400 to 430. That difference is small enough to be considered insignificant. Fig. 1 shows the number of requests per second, per compression in detail. When inserting 200 000 rows, the LZO Compression, the SNAPPY compression and the control test without compression behave

in similar manner. The GZ and LZ4 compressions are a fraction slower. The insertion of 400 000 rows is similar regardless of the chosen compression and starting from the 600 000 and upward clearer differences in the execution time and the number of requests per second can be observed. What makes an impression are the spikes in the LZO compression and the test without any compression. In the tests with 800 000 and 1 000 000 rows, the cases without compression and the LZO compression behave best, respectively.

2) Select queries: The select test cases are performed on the same tables that have been created when conducting the Insert experiments. Meaning that the select queries are performing full table scans on the compressed data. For every combination of dataset and compression, a total of three test scenarios are done – querying 25%, 50% and 75% of the data. Fig. 2 shows the execution time of the conducted tests. Almost every compression is behaving as expected – with the increase of the queried rows, the execution time increases linearly. LZO is the only compression that does not follow this behavior – it has some notable spikes when selecting 150 000, 300 000 and 450 000 rows. A possible explanation of that can be the locality of the data and it is worth a further examination.

C. Comparative Experiment on Physical Storage

The physical storage of the data (the Persistent Memory) is also an area that is worth looking into. The column-oriented systems are characterized by improved data compression. Moreover, that can be seen in Fig. 3 in which it is obvious that the tests run with no compression are taking up a lot of physical memory on the cluster. Tables with more than 1 000 000 rows can be as large as 1GB. Fig. 4 shows the physical memory of the data without the outlier – the tests with no compression. In this figure, the difference between the four compressions is more distinguishable.

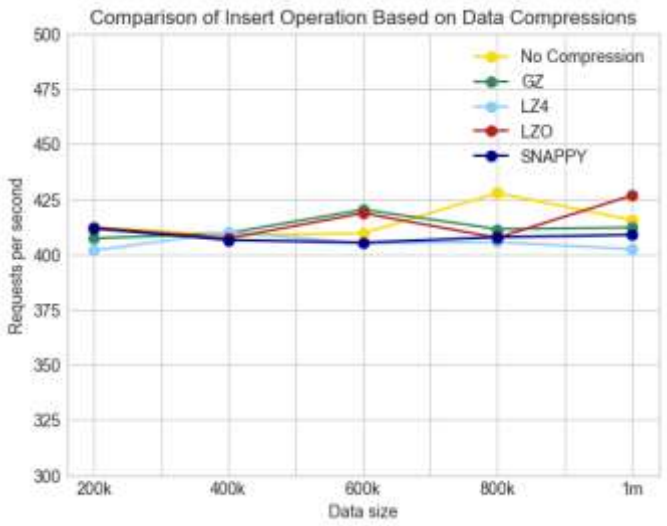

Fig. 1. Comparison of Insert Operation.

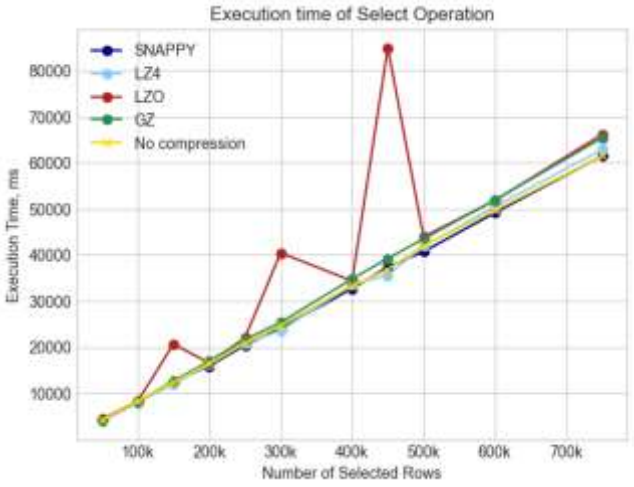

Fig. 2. Comparison of Select Queries.

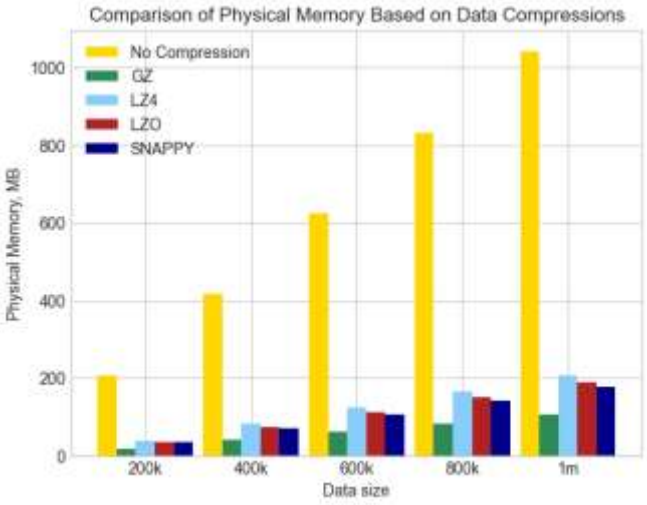

Fig. 3. Comparison of Physical Memory for all Test Cases.

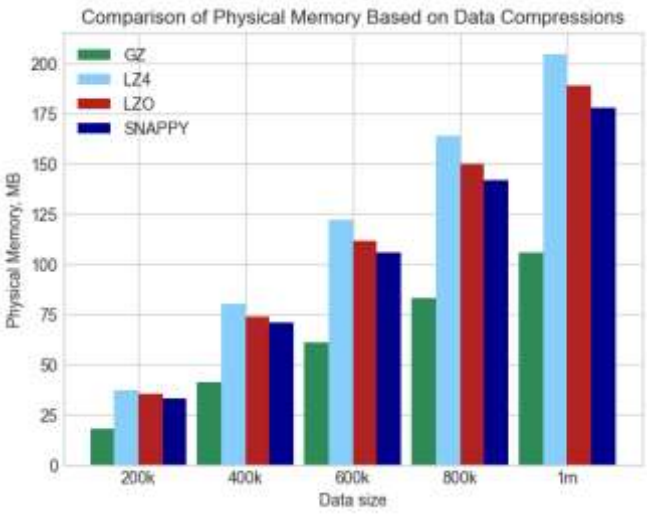

Fig. 4. Comparison of Physical Memory for the Compression Tests.

The GZ compression is clearly the most sparing of the four, storing 1 000 000 rows in little more than 100MB; in other words, 10 times less than the insertion without compression. The GZ compression is followed by SNAPPY compression and the LZ4 compression is the one that takes up most storage space.

D. Comparative Experiment on Number of RegionServers

The number of the RegionServers is another indicator that is taken into consideration. It confirms the benefit of using data compression algorithms. Although the difference in the number of RegionServers when using compression and not, is small, it does affect the whole performance of the cluster when inserting and selecting multiple records. The number of the occupied RegionServers while inserting rows without compressions is up to two for more than 600 000 inserted rows, while using any compression results in one reserved server every time.

E. Cluster CPU Load

Fig. 5 shows the load of the Cluster CPU while performing 400 000 insert queries. The left part of the figure shows the CPU load of the cluster when insert data without any compression and the right part shows the CPU with the LZO compression. Clear from the figure is that regardless of the used compression, or the lack of, the CPU is loaded the same.

Fig. 6 denotes the data transferred through the HDFS with and without any compression. The left side of the figure is the insertion of 400 000 rows of uncompressed data and the right side – the same number of rows with LZO compression. As expected, the compressed data is less and therefore the I/O operations on the HDFS are less. What makes an impression is that regardless of the data that is being transferred and the additional processed linked to the compression, the time of the whole insert operation is almost even.

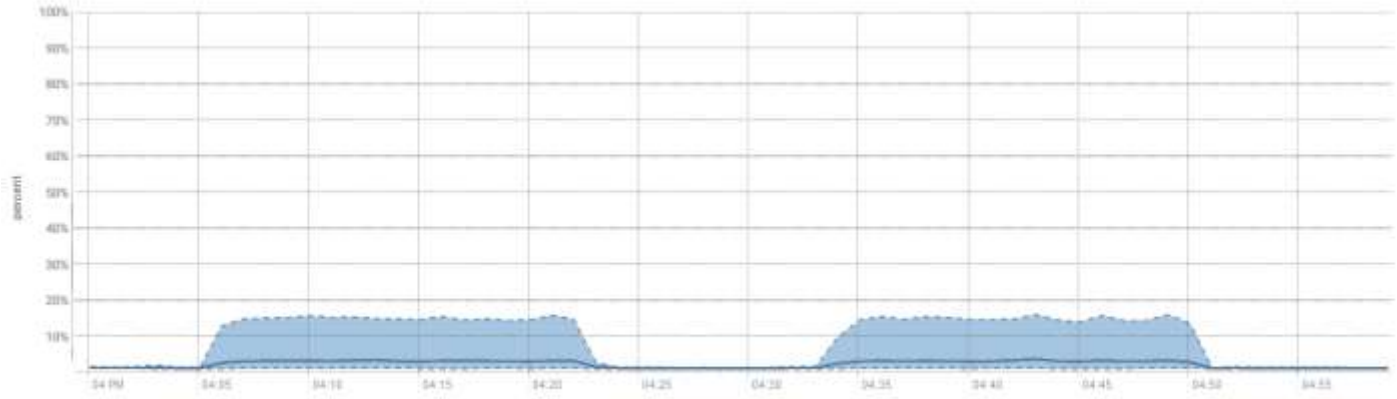

Fig. 5. Cluster CPU Load.

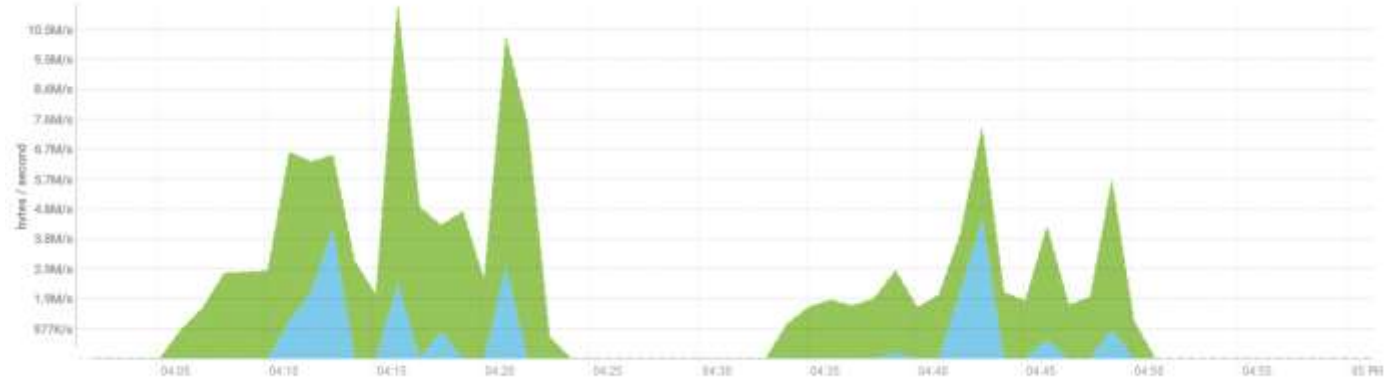

Fig. 6. HDFS I/O.

VI. Conclusions and Future Work

The aim of this research is to determine if data compression could improve the performance of HBase, running on a Hadoop cluster, consisted of one name node and nine data nodes. After performing a series of experiments, we can conclude that:

- Data compression highly improves system performance in terms of storage saving. It could shrink data 5 to 10 times (depending on the algorithm) without any noticeable additional CPU load.
- The substantial decrease in the network traffic could have a significant positive impact on large clusters with many data nodes or in clusters with higher data replication factors.
- The GZ algorithm achieves the highest compression ratio from all four applied algorithms.
- The cluster does not need any additional computational resources or CPU time to compress or decompress data, as it was expected.
- Regardless of the used compression, or the lack of it, the time for inserting and selecting thousands of rows does not change in noticeable manner. Neither the requests per second. When running the experiments, the active nodes' CPU load was less than 20%. All these measurements may suggest that there is some kind of limitation setting in the Thrift server or in the ThriftPy2 package. Further research and analysis are needed.

In future, we plan to extend our work and:

- Find a way to increase the number of processed requests per second. Obviously not the hardware is the limitation factor in the present experiments.
- Add more sources of data and simulate a parallel cluster usage from multiple clients.
- Increase the replication factor and analyze how data compression impacts the cluster performance in that case.

References

[1] F. Bajaber, S. Sakr, O. Batarfi, A. Altalhi, A. Barnawi, "Benchmarking big data systems: A survey," Computer Communications, 149, 241–251, 2020.

[2] A.H. Abed, "Big Data with Column Oriented NOSQL Database to Overcome the Drawbacks of Relational Databases," Int. J. Advanced Networking and Applications, 11(05), 4423–4428, 2020.

[3] W. Ali, M.U. Shafique, M.A. Majeed, A. Raza, "Comparison between SQL and NoSQL databases and their relationship with big data analytics," Asian Journal of Research in Computer Science, 1–10, 2019.

[4] J. Agnelo, N. Laranjeiro, J. Bernardino, "Using Orthogonal Defect Classification to characterize NoSQL database defects," Journal of Systems and Software, 159, 110451, 2020.

[5] R. Yangui, A. Nabli, F. Gargouri, "Automatic transformation of data warehouse schema to NoSQL data base: comparative study," Procedia Computer Science, 96, 255–264, 2016.

[6] D. McCreary, A. Kelly, "Making sense of NoSQL," Shelter Island: Manning, 19–20, 2014.

[7] S. Wandelt, X. Sun, U. Leser, "Column-wise compression of open relational data," Information Sciences, 457, 48–61, 2018.

[8] P. Raichand, "A short survey of data compression techniques for column oriented databases," Journal of Global Research in Computer Science, 4(7), 43–46, 2013.

[9] R. Čereš\vnák, M. Kvet, "Comparison of query performance in relational a non-relation databases," Transportation Research Procedia, 40, 170–177, 2019.

[10] J. Sun, T. Lu, "Optimization of column-oriented storage compression strategy based on HBase," in 2018 International Conference on Big Data and Artificial Intelligence (BDAI), 24–28, 2018.

[11] D. Abadi, S. Madden, M. Ferreira, "Integrating compression and execution in column-oriented database systems," in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 671–682, 2006.

[12] Raichand, P., & Aggarwal, R. R. (2013). A short survey of data compression techniques for column oriented databases. Current Trends in Information Technology, 3(2), 1-6.

[13] Wandelt, S., Sun, X., & Leser, U. (2018). Column-wise compression of open relational data. Information Sciences, 457, 48-61.

[14] Raichand, P., & Aggarwal, R. R. G. (2013). Query execution and effect of compression on nosql column oriented data-store using hadoop and hbase (Doctoral dissertation).

[15] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Transactions on Computer Systems (TOCS), 26(2), 1–26, 2008.

[16] Apache Phoenix project: http://phoenix.apache.org/.

[17] X. Li, T. Georgiou, Migrating Messenger storage to optimize performance. Facebook Engineering, 2018: https://engineering.fb.com/2018/06/26/core-data/migrating-messenger-storage-to-optimize-performance/

[18] Fang, J., Chen, J., Al-Ars, Z., Hofstee, P., & Hidders, J. (2018, September). A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress. In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (pp. 1-2).

[19] Prasanth, T., Aarthi, K., & Gunasekaran, M. (2019, April). Big Data Retrieval using HDFS with LZO Compression. In 2019 International Conference on Advances in Computing and Communication Engineering (ICACCE) (pp. 1-6). IEEE.

[20] Jiang, H., & Lin, S. J. (2020). A rolling hash algorithm and the implementation to LZ4 data compression. IEEE Access, 8, 35529-35534.

[21] Packer, C., & Holder, L. B. (2017, August). GraphZip: Mining graph streams using dictionary-based compression. In SIGKDD Workshop on Mining and Learning in Graphs (MLG).