DOI: 10.15439/2016F486
ACSIS, Vol. 8. ISSN 2300-5963

The Column-oriented Data Store Performance Considerations

Artur Nowosielski[1,2]

[1] PhD Candidate
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6, 01-447 Warsaw, Poland
Email: artnowo@gmail.com

[2] Findwise Sp. z o.o.
ul. Wspólna 35/16, 00-519 Warsaw, Poland

Piotr A. Kowalski[3,4], Piotr Kulczycki[3,4]

[3] Faculty of Physics and Applied Computer Science
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Cracow, Poland
Email: {pkowal,kulczycki}@agh.edu.pl

[4]Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6, 01-447 Warsaw, Poland
Email: {pakowal,kulczycki}@ibspan.waw.pl

***Abstract*—The massive amounts of data processed by information systems raise the importance of detailed database performance analysis. Column-oriented data stores are becoming increasingly popular in big data appliances. This paper identifies database performance factors on the basis of empirical studies on a custom implementation. To summarize the research, a simple performance mathematical model has been created.**

I. Introduction

THIS article is the result of experiments performed on a custom column-oriented database management system. Performance studies presented in this paper are a part of a broader research initiative about an optimization of columnar data store sharding with the use of a natural computing algorithms. The research has been overviewed in the previous FedCSIS conference paper. [1] Performance modeling and discovering what determines a single database's behaviour is considered as the first step towards creating a more sophisticated partitioned database model. Such a model will be a subject of optimization by metaheuristic algorithms. Columnar data store performance studies are considered important because non-relational data stores are gaining popularity and more and more applications. However, the goal of this research is not a comparative study of the column-oriented database model versus other data models, nor CODB custom implementation versus other columnar data stores. It aims to discover which factors determine performance and the relationships between these factors' impact. Absolute values are considered less important in this study since it is intended to be a foundation of the weights (importance) estimation of the specific factors.

The paper is structured as follows. Section II contains a general description of a column-oriented database model relatively to the relational model for the sake of better understanding, and the custom implementation of columnar data store is introduced. The next section III presents how the performance of a Java application (specifically a database) can be measured and expressed. The second half of the article comprises of sections IV, V and VI, which include assumptions and conclusions brought on the basis of the experiments' results.

Because the research and benchmarking is performed on a custom database management system, the article has relatively unique character. However, available literature offers examples of research initiatives driven by similar ideas, such as [2] or [3]. Since this is a short paper, it presents a general overview and the most important facts of the research.

II. Column-oriented DBMS

The column-oriented database model has been around for almost as many years as the most popular, row-oriented relational model by E.F. Codd [4]. One of the first concepts regarding column-oriented storage are transposed files databases from late 1970s [5].

After an initial rush, columnar databases remained in their own, narrow niche for more than 20 years, while the relational model dominated a majority of applications. Relational model's strength came from the strong mathematical foundation based on the set algebra and focus on the data consistency and reliability. Despite this, in the recent decade, alternative models have gained more interest in the commercial world with the rise of the non-relational database trend. Databases that focus on other aspects than traditional Relational Database Management Systems (RDBMS) started gaining more attention. This trend has been called the *not only SQL* or *NoSQL* and was one of the outcomes of a rise of interactive, especially social, web services within the *web 2.0* movement. [6] The most significant developments in the area of columnar data stores are the C-Store [2] [7] and MonetDB [7].

Viewed from some angles, it can be said the column-oriented model is essentially only a physical-tier modification of the relational model. [3] However, experiments prove that implementing adequate modifications to the row-oriented Database Management System (DBMS) storage tier is not enough to rival the columnar store in some applications [8]. Nevertheless, some column database management systems offer roughly the same interface as the relational ones, hiding all the internal differences [9]. Certainly, there are analogies between fundamental terms of both domains. Keyspace parallels a database from the RDBMS domain. Relational table (or relation) is roughly the same as a column family in columnar stores. Correlation of values with the same key from different columns in columnar store leads to construction of a tuple, which is comparative to a record in the relational database.

Despite obvious analogies, there is a fundamental difference in the storage structure. In row-oriented stores there is a single data file for the whole table and data is stored in a tuple-by-tuple manner where tuples are stored column-by-column. Thanks to this, it is cheap to iterate over records and read all the values in order to construct a complete tuple. Moreover, it is cheap to write the whole record at once by just appending or replacing another entry. However, this architecture has significant disadvantages too. Firstly, the table schema is effectively immutable - changing the column set in a table requires rebuilding the whole data file and all of the indices. Secondly, iterating through values in a single column from all the records requires long jumps within a data file, which involves huge I/O overhead.

These issues are addressed by column-oriented architecture. In a classical approach with explicit record ID storage [7], each column is a sequence of id-value pairs stored in a separate file. Database schema is flexible, columns can be freely added or removed and if a given record does not have value in a given column, it does not take any space to record such an information. Iteration over every value in a column is simple, so that columnar stores are effective in terms of generating aggregations, summaries and other read-intensive applications [3] [9]. But this schema also has a significant drawback. It is expensive to collect all the distinctive elements of a record altogether. Such an operation requires searching of all the data types (columns).

Taking this into consideration, in a domain of columnar data stores, a concept of the record does not actually exists. At the lower tiers of the system, there is no such concept and a specific values in a *record* can be associated to others only by matching the row id. This is the most fundamental conceptual difference between the two models. The columnar model storage is organized around pieces of information of the same type, not of the same entity.

This paper relies solely on the CODB database management system. The implementation has been partially reviewed in [1]. Database logic is comprised of execution of a set of processes touching a physical storage tier. A given part of an algorithm is considered as a significant in terms of performance when its execution time or resource consumption depends on any external factor. For example, looking for a value in the value storage file depends on a number of currently stored values in the set, whereas appending new value in some cases does not, in case when it is performed at EOF (end of file - the very last possible offset in a file), given that the EOF can be obtained instantly from the filesystem. In order to decrease space waste and speedup operations, a concept of storage maps has been implemented. Storage maps are responsible for keeping track of free space chunks. When any operation needs to allocate a bit of space, it asks the storage map first, and jumps to EOF only if free space does not offer an adequate fragment. For the same reason, a key storage data file operates on value hashes instead of actual values. Hashing overhead is orders of magnitude lesser than the potential overhead caused by moving around bigger pieces of data.

III. Performance analysis

A very important aspect of a performance measurement is an overhead. In conformity with common sense and intuition, it must be predictable and have a minimal possible impact on the measurement's result.

Another critical concern, when it comes to software performance measurement, is concurrency and parallelism. In a classical, single-threaded sequential program execution, the matter is trivial. The execution time is proportional to a cycle per instruction (CPI) value, whereas CPI is an inversion of the instructions per cycle (IPC) value with constant, known cycle time. [10] In multi-threaded or parallel conditions none of these assumptions are true. At the time of writing this paper, the test CODB system operates in a single thread for the most of the time. The only multi-threaded parts are Java parallel streams used for processing some of the internal collections. This does not affect database logic, which is discussed in this paper. Taking this into consideration, sequential processing measurement techniques were used.

There are many execution parameters which can be measured. For the sake of the research, the following parameters were chosen: execution time, CPU workload and a heap size. Such a selection lets us to take two important perspectives of the system's performance: the user view (time-oriented) and the system view (resource-oriented). The user perspective is connected to considering system as efficient, it determines system's capacity as well. The faster requests are processed, the more of them can be handled in a unit of time. This aspect is particularly important in interactive applications, requiring fast responses for a massive amount of requests. A fundamental time metric is the execution time. The other perspective, a system one, is resource-oriented. Resource is a part of a system, which serves for the other elements of the same system. [11]

Metric is defined as a way to determine whether a system has given property or not and to what extent. Specifically, in the performance engineering domain, metrics provide information about performance parameters with regards to time and amount of computational work. Application context is crucial for interpretation of a metric's result. For time-sensitive applications, like real-time systems, time domain is fundamental, whereas applications processing huge volumes of data will put a pressure on throughput, regardless of resource usage.

In the literature of the subject there is no generally-adopted standard on performance metrics. They are rather ad hoc, defined for each application or class of applications. However, there are common types of metrics used, compliant with two major perspectives mentioned earlier. In this research, the following metrics have been used: Response time [ms] - the total time of a request execution; Throughput (capacity) - the number of requests completed in a given time unit; Resource consumption [MB] - the highest JVM heap size during the test execution. In terms of scalability, the most important metric is the capacity.

A custom, configurable workload generator has been used. It enabled the configuration of the following aspects of the generated requests:

- read/write requests ratio;
- data granularity: defined or random size of entries;
- data entropy: a number of generated values or a unlimited randomness;
- whether the generated requests should touch many columns or not;

CODB Benchmark application configures a dedicated loggers for the sake of collecting execution time, throughput and heap measurements. Measurement data is written in CSV (comma-separated values). Execution time is calculated as a difference between two consecutive `System.nanoTime()` invocations, one just before measured method invocation and one just after. JVM heap consumption is measured using the `java.lang.Runtime` class.

IV. Performance factors

This section presents a set of factors which are suspected to impact on database performance. They were chosen based on expertise, but do not necessarily have a real, significant influence. Verification of these suspicions is the key objective of benchmarking.

First category of performance factors are the universal factors, related to data processed by a database, but unrelated to specific technology concerns.

Read and write operations ratio can determine a lot of performance-impacting elements. Firstly, write operations require synchronization effort. For obvious reasons, insert, update or delete requests imply I/O operations, which need to be enqueued and buffered on many tiers of the system down to the physical layer, where they are really executed.

Assuming a constant amount of data to be written; fewer, big portions are expected to be processed faster than a larger number of small portions. With the use of a benchmark, it needs to be measured to what extent such a prediction is true.

Due to CODB storage file structure, which is essentially a RLE-like-compressed structure, entropy is predicted to have a significant influence on performance of the system. RLE, or run-length encoding, is a simple way of data compression based on substitution of numerous occurrences of a term with a single instance and a number value which represents the number of occurrences. The lesser the entropy is, the better performance should be, because as stated before, when a single column is discussed, appending a new entry with an already existing value may require as little as increasing a single 8B counter and writing a 16B key.

Requests are issued by multiple threads, but the threshold on which synchronization and context switching between threads becomes bigger than concurrency performance gain is unknown. Benchmarking may provide a reasonable empirical information regarding how many threads is too many and, specifically, how that number is related to the CPU's core number and the CPU's hyper threading capabilities.

Besides the technology-independent factors, performance can be affected by technology-specific components. From the wide variety of candidates, two were chosen for the research as potentially having the biggest impact on results.

The Java Virtual Machine platform, and thus the Java programming language, memory model is based on indirect memory management. The application does not allocate and release the memory occupied by the objects on its own, but it is done implicitly by the JVM within a part of the memory called the heap. A critical component of the memory management facility in Java is the garbage collector, a module responsible for removing unused objects from heap. According to the official documentation [12], the HotSpot JVM v.1.8 provides four garbage collector implementations: serial, parallel, concurrent mark-sweep (CMS) and G1.

As CODB is executed on the JVM, it's state may (or may not) have an impact on performance. For example, some internal data collections or buffers are expanded exponentially, so that at the beginning (*cold* state) it will happen more frequently than later (*hot* VM). Hot tests are performed by issuing 1000 write/read requests before starting measurement. After the warm-up all the database internal structures are cleared in order to avoid performance impact by having prefilled collections or buffers.

V. Results and discussion

Each test was performed 4 times and consisted of issuing 50 000 requests. Tables I, II and III present the measurement results. In all the tables, extreme values which are to be discussed further are highlighted. Every table header contains information about what values are desired (low or high).

Each section in result tables contains results with different values of a single factor. Unless a given factor is tested, the following values were stated as defaults for each test: parallel garbage collector, hot JVM, r/w ratio = 0.5, low entropy and two columns in use.

Testing environment was: Intel Core i7-4600U CPU with 12GB DDR3-1600 RAM and SSD drive with the ext4 filesystem. The operating system was a 64-bit GNU/Linux 4.2.0 with Oracle HotSpot 64-bit JVM v1.8.0-74. Benchmark was started using Maven Exec Plugin.

Garbage collector implementation has very low impact on request execution time. Results for all the implementations are similar and of similar stability (almost the same standard deviation). Serial GC performed the best, probably because of relatively low data volume and single threaded testing. Stopping a single thread is less harmful in terms of performance than stopping multiple threads. For heap usage levels GC has an obviously fundamental impact, although some patterns are visible here. Parallel GC achieved the lowest minimum heap size of as little as 15MB. This result probably is an outlier, because of distance from all the other implementations and needs a deeper investigation. The highest standard deviation also may be skewed by one or more outliers. Differences in terms of

TABLE I
BENCHMARKING RESULTS: REQUEST EXECUTION TIME (THE LOWER THE BETTER)

Factor	Name	Min[ms]	Max[ms]	Avg[ms]	Std deviation	Impact
GC type	serial	0.004	186.739	0.060	0.483	low
	parallel	0.004	189.110	0.067	0.654	
	CMS	0.004	188.975	0.059	0.485	
	G1	0.004	188.600	0.059	0.633	
JVM state	hot	0.004	189.110	0.067	0.654	immaterial
	cold	0.004	189.188	0.0623	0.816	
Read/write ratio	only reads	0.002	**11.171**	**0.006**	0.036	high
	50% writes	0.004	189.110	0.067	0.654	
	only writes	**0.027**	**189.935**	**0.087**	0.728	
Data entropy	low	0.004	189.110	0.067	0.654	high
	high	**0.001**	**30.246**	0.060	0.306	
Multicolumn	yes	0.004	189.110	0.067	0.654	low
	no	0.004	183.657	0.062	0.480	

TABLE II
BENCHMARKING RESULTS: HEAP USAGE (THE LOWER THE BETTER)

Factor	Name	Min[MB]	Max[MB]	Avg[MB]	Std deviation	Impact
GC type	serial	29.50	**425.52**	116.16	81.96	high
	parallel	**14.98**	417.96	124.32	**103.20**	
	CMS	28.97	347.42	**141.60**	76.67	
	G1	30.25	266.02	135.67	64.61	
JVM state	hot	14.98	417.96	124.32	103.20	high
	cold	17.73	236.70	93.38	62.57	
Read/write ratio	only reads	**61.77**	125.05	93.40	**28.80**	high
	50% writes	14.98	**417.96**	124.32	103.20	
	only writes	17.04	373.58	139.91	97.91	
Data entropy	low	14.98	417.96	124.32	103.20	high
	high	14.12	**142.84**	**58.10**	**48.46**	
Multicolumn	yes	14.98	417.96	124.32	103.20	high
	no	14.89	**300.54**	**89.75**	67.85	

TABLE III
BENCHMARKING RESULTS: THROUGHPUT (THE HIGHER THE BETTER)

Factor	Name	Min[req/sec]	Max[req/sec]	Avg[req/sec]	Std deviation	Impact
GC type	serial	**10518**	18237	12771	2373	moderate
	parallel	**6986**	17600	11475	3125	
	CMS	8492	18608	12891	2707	
	G1	8834	**18924**	**13286**	2825	
JVM state	hot	6986	17600	11475	3125	low
	cold	5920	18975	12095	3380	
Read/write ratio	only reads	**41400**	**50000**	**47850**	3724	high
	50% writes	6986	17600	11475	3125	
	only writes	**5491**	**14693**	**9617**	3153	
Data entropy	low	6986	17600	11475	3125	low
	high	**10455**	**13940**	**12045**	**1337**	
Multicolumn	yes	6986	17600	11475	3125	low
	no	**8699**	16381	**12277**	2708	

maximum and average recorded usage are much more stable. Execution with the serial GC consumed the highest amount of memory, which seems to be a trade-off of its simplicity and speed. In terms of request processing throughput, Serial GC has the best minimum recorded throughput, but the true winner is the G1 GC, which offered the highest maximum result and the highest average throughput. Results are similar and the impact is relatively low, though.

The JVM state variable shows a little impact on performance in both time-oriented metrics. It has much higher impact on the heap usage levels, but that would probably make sense - the longer the program is running, the higher is heap usage. In connection with similarly low standard deviation, it brings a conclusion that the JVM state is not very important for the performance. In general, technology-specific factors turned out to have much less impact on a database performance.

When it comes to the data oriented metrics, their impact is much more visible in performance results. Read-write ratio, according to intuition, showed that read operations are performed much faster than write operations. A clearly visible pattern is present in both time-oriented metrics. Starting from the 50% share of write operations, results are stable. This may represent the logarithmic dependency. In terms of heap usage, 100% read pattern showed surprisingly high minimum recorded usage, which may be an outlier. The highest maximum usage was registered for 50-50 pattern. Probably the pattern here is the higher diversity of objects, the higher memory usage is. This requires a further investigation, but is not very important in the research context. Relatively low standard deviation in time-oriented metrics shows that results are quite stable.

Data entropy presents very interesting case in the request execution time results. The request processed in 0.001ms may be a previously discussed corner case with the insertion of a new instance of already existing value that requires to only append 16B and increment a single long variable. Enriching the execution time log with additional information about request type would help to verify this prediction. High entropy resulted also in a very good maximum request processing time. The reason is probably again an appending at EOF. High entropy tests also proved to have much more stable result than low entropy tests. In terms of heap usage, pattern is similar, high entropy has much better and much more stable results. Throughput metric repeats discoveries from request execution time, as these metrics are related to each other, in a sense. The entropy results are surprising, and require a deeper data mining in order to bring more specific conclusions on its impact.

The last analyzed variable was a multi-column vs. single-column mode. Single column mode showed better results in all of the metrics, although the impact on request execution time results is relatively low. Influence on throughput is moderate, whereas the highest impact is put on the heap usage levels. Standard deviations are similar, but relatively high.

In the context of a horizontal scalability, especially important for this research, data-oriented observations are viable. An optimum case of a single-column with a high entropy and a relatively low share of write operations emerges from the results.

VI. Performance model

In order to produce the mathematical model, chosen factors presented in sections IV and V needs to be converted to a mathematical value. The model is necessary to estimate a performance of a database instance with the specific parameters. This is the first approach to a simplified model and it is to be refined in subsequent work. Weights are determined on the basis of impact displayed in tables I and III. Weights sum must be equal to 1. Other variables were considered immaterial and thus are not present in the formula. Garbage collector implementation also was skipped, as its impact on time-oriented metrics is much lower than data-oriented factors. P metric estimates database instance performance, the higher is the better.

$$P = 0.2 * MC + 0.4 * E + 0.4 * \frac{1}{ln(RW + 1.1)} \tag{1}$$

MC represents a single- or multi-columnar mode. At the moment it is defined as a binary factor with values of 0 for a multicolumn, and 1 for a single column mode. In future research the MC factor may need to be refined to a functional form, depending on the number of columns involved. E takes a values from range $[0, 1]$ where 0 is using the same value all the time and 1 means a total randomness. RW is a ratio between the read and write operations, in range $[0, 1]$ where 0 is the read only and 1 write only. Taking everything into consideration, P metric can take values from range approx $[1.35, 4.80]$. The lowest value is a write-only, multi-column instance with a low entropy, whereas the highest is achieved for a single-column read-only instance with a high entropy. This model will be validated and enhanced during the further work.

VII. Summary

This paper is the very first phase of a performance analysis of column-oriented database management system. Column-oriented databases were described in details, in relation to the popular relational model. Some of the CODB implementation details were presented, putting special emphasis on the data layout. Then, performance engineering concerns were reviewed along with performance metrics. Consecutively, different components, both technological and data-originated, with potential influence on performance results were discussed and verified. Finally, a first approach to a database mathematical performance model was created and discussed, on the basis of the results.

References

[1] A. Nowosielski, P. A. Kowalski, and P. Kulczycki, "The column-oriented database partitioning optimization based on the natural computing algorithms," in *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Łódź, Poland, September 13-16, 2015*, 2015. doi: 10.15439/2015F262 pp. 1035–1041. [Online]. Available: http://dx.doi.org/10.15439/2015F262

[2] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," *VLDB Conference*, pp. 553–564, 2005. doi: 10.1007/BF02443652. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083658

[3] A. Lübcke, "Challenges in workload analyses for column and row stores," *CEUR Workshop Proceedings*, vol. 581, pp. 5–8, 2010.

[4] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 26, no. 6, pp. 64–69, 1983. doi: 10.1145/357980.358007. [Online]. Available: http://dx.doi.org/10.1145/357980.358007

[5] P. Svensson, "On search performance for conjunctive queries in compressed, fully transposed ordered files," in *Very Large Data Bases, 1979. Fifth International Conference on.* IEEE, 1979, pp. 155–163.

[6] K. Grolinger, W. a. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, p. 22, 2013. doi: 10.1186/2192-113X-2-22. [Online]. Available: http://www.journalofcloudcomputing.com/content/2/1/22

[7] D. Abadi, "The Design and Implementation of Modern Column-Oriented Database Systems," *Foundations and Trends{®} in Databases*, vol. 5, no. 3, pp. 197–280, 2012. doi: 10.1561/1900000024. [Online]. Available: http://dx.doi.org/10.1561/1900000024

[8] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?" *Sigmod*, vol. June 9-12, pp. 967–980, 2008. doi: 10.1145/1376616.1376712

[9] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *Proceedings - International Conference on Data Engineering*, 2007. doi: 10.1109/ICDE.2007.367892. ISBN 1424408032. ISSN 10844627 pp. 466–475.

[10] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008. doi: 10.1109/MM.2008.44

[11] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Future of Software Engineering (FOSE '07)*, 2007. doi: 10.1109/FOSE.2007.32. ISBN 0-7695-2829-5 pp. 171–187. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4221619

[12] Oracle, "Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide," 2016.