Database Storage Format for High performance Analytics of Immutable Data

Mikhail M. Rovnyagin[1], Sviatoslav O. Dmitriev[2], Hrapov Alexander S[3], Maksutov Artem A[4], Turovskiy Igor A[5]
Alexander S. Hrapov, Artem A. Maksutov,
Igor A. Turovskiy
National Research Nuclear University "MEPhI"
Kashirskoye highway 31, 115409,
Moscow, Russian Federation
[1]m.rovnyagin.2015@ieee.org, [2]sodmitriev96@gmail.com, [3]arcanius@list.ru, [4]aamaksutov@mephi.ru, [5]i.turovskiy@yandex.ru

***Abstract*— Most of modern database management systems offer a set of data manipulation operations, which strictly limits the available methods of data storage optimization. This article describes a database storage format that provides a low latency access to stored data with highly optimized sequential data extraction process by prohibiting any data modification after initially loading the data. The current study is aimed at developing a database management system that is suitable for high performance analytics of immutable data and performs better than database management systems with wider applicability. This paper includes developed data storage formats, data load and extraction algorithms and performance measurements.**

Keywords—databases; data storage; data analytics; NoSQL; database index

I. Introduction

Data analytics is one of the most time-consuming tasks in modern computer science. An essential part of any data analytical process is a choice of correct data storage system.

NoSQL systems are currently the most valid solution for big data storage throughout analytical process. Apache Cassandra can be given as an example of commonly used NoSQL DBMS that are used for data analytics purposes. Huge corporations who benefit from a high performance data analysis apply several requirements for database systems they use. Those requirements include high data rate and scalability [1]. In this work, we propose a database format that offers an increased data search and extraction rate and is sufficiently scalable when working with historical data.

While analyzed data sometimes may be acquired during the analytical process (e.g. analysis of user activity over the internet), it is much more common to analyze historical data. An important feature of historical data is that it is partially or totally immutable. It is common for historical data to be expanded with additional pieces of information, however it is much less common for the data to be modified after it was added to the storage.

Most of the databases that are aimed towards data storage for analytical process provide an efficient way to modify previously stored data, which directly affects their storage format by limiting possible optimization methods and thus reducing efficiency of data extraction. In-particular, commonly used Apache Cassandra sets an efficient data modification on a highly distributed database as its primal objective [2]. As a result, many adjustments were made for this DBMS. Those adjustments offer a great scalability and efficiency of data modification, but limit data extraction efficiency. This problem may affect many categories of data analysis, starting with statistical data analysis and ending with machine learning.

The database storage format proposed herein offers direct optimizations that are aimed towards immutable data storage, but the proposed DBMS may deteriorate in performance when used to store data that is often modified. The developed format not only provides efficient sequential data extraction and extraction by id, but also requires less hard drive memory to store data than conventional DBMS. Performance and storage efficiency of the developed system were determined experimentally. Results of the experiment are presented herein.

II. Conventional Storage Format

Database file layout varies for different DBMS. In this section, we present a database file layout of PostgreSQL DBMS, as it is one of the most commonly used and well documented DBMS. While the exact storage structure may vary for different systems, the overall approach remains the same. Furthermore, we will only review relational DBMS in this section as the most widely used database category. Database files most commonly represent a fully operational file system, where different directories correspond to different tables and different files to different types of data. While this file system aspect is essential for any DBMS, it will not be further explored herein, because our proposal does not include any modifications to it. Instead we will focus on the format of table and index files.

Both table and index files are usually stored identically. A table file consists of a set of pages. Each page has the same size and includes a header, an array of <offset, length> pairs, pointing to the database objects, such as table rows or index nodes, and an array of those objects. The described page layout is shown on Fig.1.

978-0-7381-4275-3/21/$31.00 ©2021 IEEE

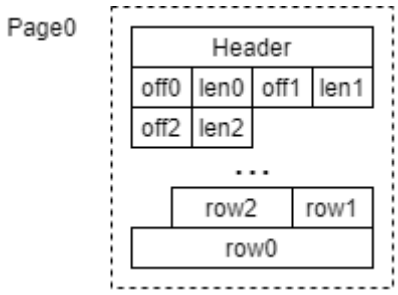

Fig. 1. Page layout of table files

To add an additional data object, a pair of offset and length must be appended to the beginning of the page, and object data must be appended to the end. If all pages are full, an additional page should be created. The database file format permits page addition to the middle of the file to preserve ordering.

For relatively fast data access, conventional DBMS provide indixes. An index allows DBMS to calculate a row offset in table file based on a value that is contained within that row. Two types of indexes exist: cluster index and non-cluster index.

Cluster indexes work by sorting all rows within the same table by one of their values (e.g. primary key in RDBMS). To find a row by cluster index DBMS can simply perform a binary search within the table file. This operation is performed with O(logN) complexity, which is not ideal. It is more than usual to use an auto incrementing id as a primary key for a row, in which case the row storage could be optimized to provide a way to find a specific row with O(1) complexity. While this may not seem to provide any significant benefits when extracting data from a single table, the advantages of this optimization increase when extracting data from multiple tables and performing table joins in the process.

Non-cluster indexes are performed via associative structures, such as trees and hash-tables. T-trees and B-trees are often used as indexing trees. T-trees may often outperform B-trees in main memory databases, however B-trees are more efficient for indexing data that is stored in disk memory [3]. This is due to an ability to adjust a size of a B-tree node, that allows DBMS to efficiently utilize block storage devices.

B+ trees with an order of multiple thousands are most commonly used for indexing data in databases that store data on external memory devices. Using a great tree order helps to reduce the amount of file system operations while relying on the block-based structure of disk drives [4]. Using a B+ tree instead of a B-tree enables a fast iteration over a range of rows, for example when extracting all values less than a specified number [5]. B-trees enable a search with complexity of O(logN). In some cases, non-cluster indexes may be preferred over cluster indexes because search with B-tree involves much less file system operations than binary search. While we do not propose any way to improve non-cluster indexes herein, we offer a variety of optimizations for the storage format of indexes that are aimed towards increasing search speed and decreasing disk memory usage.

III. Database File Format

The file format proposed herein can be described as a contiguous sequence of rows of same length. Each row consists of one or multiple cells. Length of the same column within different rows is always the same, but length of different cells within the same row may vary. Conceptually, the developed format represents a simple array of objects. The proposed database file layout is shown on Fig. 2.

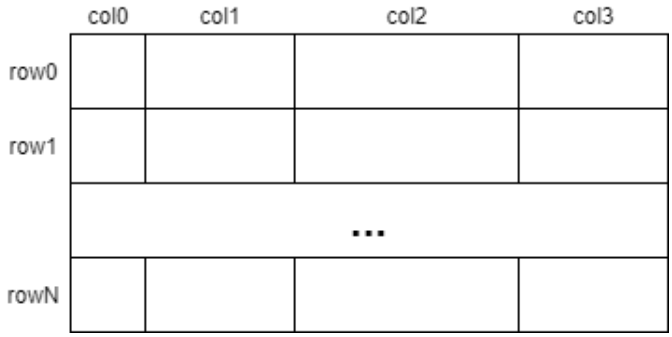

Fig. 2. Database file leyout

Simplicity of the proposed layout attributes to its fast sequential reading speed. Strict data alignment permits DBMS to extract any row by its sequential number within constant time. Furthermore, DBMS with presented data structure do not need to create a column to store an auto incrementing primary key. Instead, row sequential number can be used as a numerical row identification digit or primary key.

Lastly, database files have a header with metadata such as length and type of each column. The header also includes additional free space that should be used to store metadata specific to data type that is stored in the table. For example, indexes can store their index type in this free space. To increase data access speed, first row should be aligned to the beginning of the second block in a database file according to file system preferences.

IV. Table Storage

Database table can easily be stored in the file with proposed layout. However, few problems should still be addressed. Multiple value types can be stored in the database:

- short values whose length is lower or equal to the length of a data type that represents file offset (usually 8 bytes);
- long values whose length is higher than the length of a data type that represents file offset;
- variable-length values.

Storage of short values is straightforward. Each value is stored in a corresponding cell in the database file.

Storage of variable-length data is less obvious. Cells in the proposed database format always have constant length, so storing variable-length data directly in the table is not possible. A prominent example of a variable-length data object is BLOB (binary large object). BLOBs represent a sequence of data that

is unstructured from databases perspective. Another example is a text string. Text string are usually different from BLOBs, because a string type can usually be limited in length while BLOBs cannot.

To store variable-length data we propose using a separate unaligned file. Storing BLOBs in files that do not belong to databases file system promises an increase in data access speed [6]. Therefore, using an external file for variable-length data should increase an average data access speed.

One external file should be created either for every table or for every variable-length column. Storing all variable-length data of same table in one file seems more reasonable since it actively exploits file system cache, but in some cases creating a single data file for each column may happen to be more efficient. In addition to data itself, length of a stored value should also be stored in the variable-length data file. The proposed approach is presented on Fig. 3.

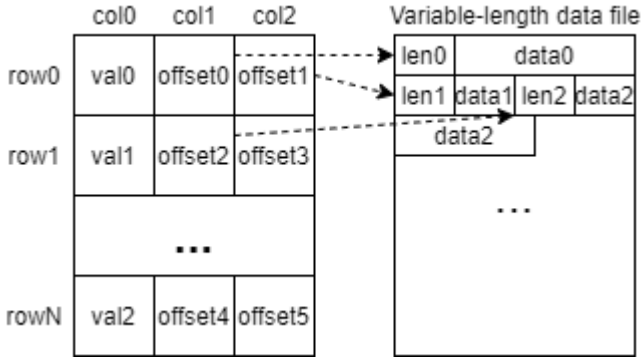

Fig. 3. Variable-length value storage

The proposed approach does not apply any restrictions a type of stored values. Both text and binary data will be stored equally efficient. However, using an external file to store pieces of data can negatively affect performance, so variable length values should be avoided during database planning, similar to conventional DBMS guidelines.

The proposed variable-length storage format can also be used to deduplicate data. If a data set includes a column with a variable size but a limited variety of possible values, each unique value will need to be stored in the database only once, while all rows that contain the same value will store an equal offset value. This concept is presented on fig 4.

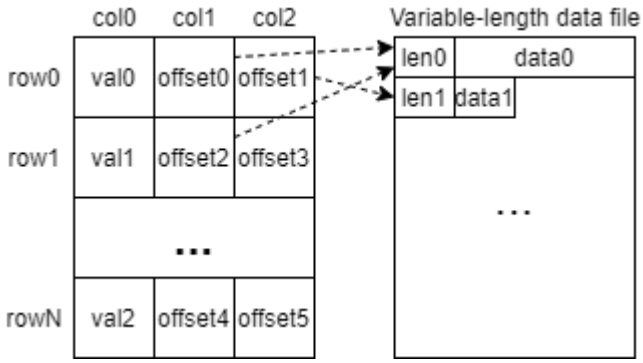

Fig. 4. Data deduplication

On figure 4 rows 0 and 1 store the same value in column 1, so they both contain the same offset, while the value itself is stored only once. While this approach reduces database file size, it also drastically complicates data addition and indexing, so it should only be used when appropriate data is present.

Long values belong to a peculiar category, as they can be stored using any of the presented methods. On the one hand, they can be stored the same way as short values to reduce complexity, but on the other hand, they can be stored as variable-length values to avoid duplication. Choice between those two storage methods should be done in accordance to characteristics of data and system limitations.

V. Index Storage

Index is a data structure that enables DBMS to find an object by one of its characteristics. Indexes are usually stored within the same data structure as tables. Previously proposed table format offers many benefits for indexes stored within it. Two index types for the proposed database format were developed: B+ tree and Aho–Corasick tree. In addition, sequential search can always be used to filter rows by a non-indexed column, albeit very inefficiently.

B+ trees can be used to filter rows that contain either an exact value or a value from a specified range. To store B-trees according to the proposed file layout each node should be represented as a row. Each row contains the following values:

- number of keys;
- id (row number) of the first child node;
- array of keys.

Array of keys can be stored in a variety of ways:

- in a single row (because maximum array size is known);
- spread across multiple rows to avoid excessive data usage;
- in a variable-length data file (along with key count).

Each way has its own benefits. However, in our experiments we used a variable-length data file to store those arrays because it is the most memory efficient approach and does not require to spread a single node across more than one row. On the downside, it results in necessity to perform three file system operations per row (read row with offset, read length, read array) instead of one.

Considering that each non-root node in B-tree is always at least half-full, nodes can be stored in a row format to increase access speed at a price of increased disk memory consumption. However, we do not recommend this optimization because a slight gain at data access speed is insignificant when the optimized process is repeated only O(logN) times.

As stated before, each stored node references only its first child. This is possible because all nodes are located sequentially within the file in the breadth-first order. In that situation, to find n'th child (starting from 0) n should be added to the id of a first child. This approach also enables an easy iteration over sequent elements, as all leaf nodes are stored

sequentially. According to this ordering, root node will always be stored in the first row and all leaf nodes will be stored at the end of the file.

Leaf nodes have a different format. Leafs can be distinguished from non-leaf nodes by a zero value in a child id column. Along with keys, they also include ids of rows with searched value. In general more than one row can have the same value. Due to that fact length of a leaf node cannot be limited, and leaf nodes have to store their arrays in a variable-length data files.

Example of tree storage in case of B+ tree of order 3 is illustrated on Fig. 5. For clearance, on figure 5 tree is represented in a non-rectangular table. On practice, same structure is accomplished by using a proposed variable-length data file format.

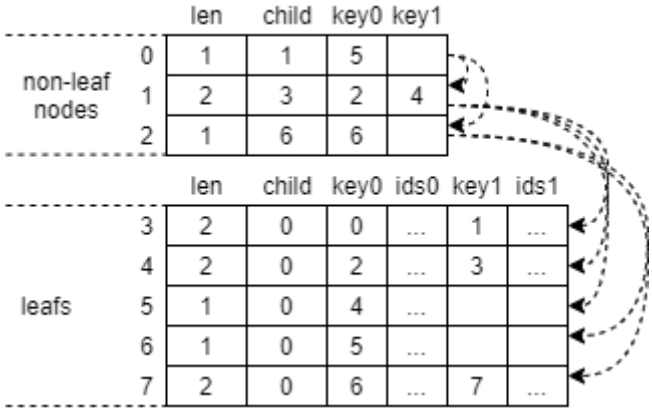

Fig. 5. B-tree storage format

Aho-Corasick tree is used to find a string or a binary vector by a regular expression. It does not perform nearly as well as B+ tree, but may be necessary in some cases to perform a versatile search. Unlike B-trees, Aho-Corasick trees greatly benefit from using parallel algorithms [7]. Firstly, this algorithm can be paralleled on node level (i.e. distribute nodes over parallel threads). Secondly, string matching algorithm used within can be paralleled as well [8].

Aho–Corasick tree can be stored similarly to B-tree, except it will not be balanced like B-tree and thus leafs will appear not only at the end of the table but in the middle as well.

VI. Data Addition

The proposed database structure is highly optimized for fast data extraction and sequential access but does not provide any way to add additional data without rebuilding the whole table. While adding data to a table is as simple as appending an additional row, adding a new value to the index is a much bigger issue. Because of index strict ordering, it is impossible to add a value to the existing index. Instead of that whole index must be rebuild to add each value. To solve this problem we suggest that new data should be loaded in chunks. In many cases it is not crucial to analyze new data that was only added a few minutes prior. By accumulating all incoming data for some period of time and then loading all accumulated data based on schedule (e.g. each night or weekend) we can bypass this limitation while not losing in data access speed. To implement such accumulator any DBMS with mutable data or a simple data log can be used. In future, we plan to reduce the described effect by implementing an efficient way to distribute the proposed database system.

VII. Results

During this study a database storage format with fast access to immutable data was developed. A prototype DBMS based on proposed approaches was created. The prototype DBMS has a distributed nature, however it lacks in algorithm optimization and may lose a significant amount of time for communication over network. The developed DBMS was tested on various data sets of different size.

Database size was measured for different data sets. Measured database files were not compressed to acquire the most representative view of disk memory usage. Testing data sets consisted of 43 columns and a varied amount of rows. An index was built for each column. Dependency of database size based on raw data size is shown on figure 6.

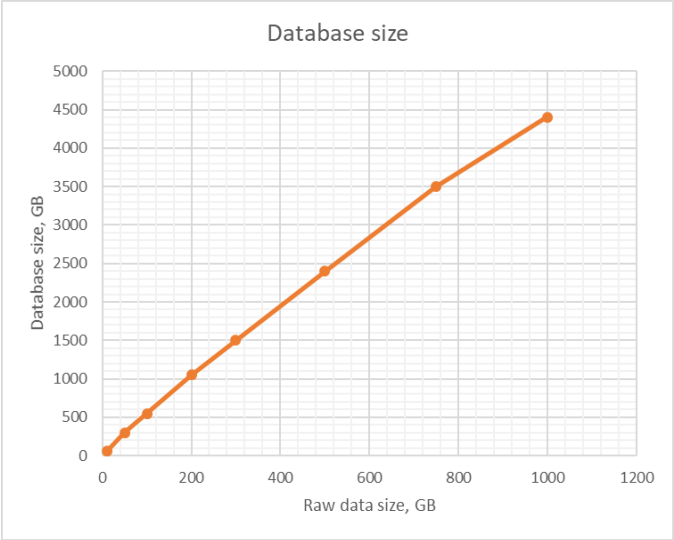

Fig. 6. B-tree storage format

According to measurements, each indexed column takes approximately 3.5% of raw data size or 150% of single row size, which is a good result considering testing data mostly consisted of integers and short strings.

Performance of the developed prototype was measured. According to measurements, developed prototype performs up to 167 index searches per process in 1 second on fully indexed 300GB dataset. Those results however are highly dependent on data, computer system characteristics and algorithm implementation. Index rebuilding with prototype system can be performed at a rate of up to 90 MB/s, depending on system characteristics.

Overall, the developed DBMS showed decent results. It may be outperformed by modern solutions due to a lack of thorough optimization and proper distribution mechanism. However, further development should improve systems performance and make it a viable competitor for modern DBMS.

VIII. Future Development

A database storage format for efficient storage of immutable data and a DBMS that uses this format were described in this paper. In future, we plan to extend the proposed database system by implementing means of distributed database storage, data encryption, compression and applying GPGPU and in-memory technologies to accelerate data search and database file compilation.

Database distribution is an effective approach to increase not only data search and extraction speed, but also new data addition. According to ideas that were presented in the previous section, new data can be periodically uploaded to the main database by entirely rebuilding indexes. If more than one data node is present in the database system, all additional data can be uploaded to a single node that is chosen by some algorithm. In this way, data upload speed will increase linearly with increasing number of nodes as long as amount of new data is far less than an average amount of data stored in a single node. Additionally, proposed database format introduces no limitations for data distribution. Classical distributed environment consists of a single control node and multiple storage nodes [9]. The biggest problem in this model is distributing data across multiple nodes in a way that allows server to evenly distribute the load across numerous nodes. Due to a strict format of a proposed database format, such data distribution can be performed mostly without limitations.

Data encryption and compression are two valid ways to increase security and efficiency of the proposed system. Both encryption and compression can be efficiently implemented due to the fact that the proposed DBMS never needs to insert additional data in the middle of a file.

GPGPU and in-memory technologies are efficient for fast data search with an index. Hybrid computer technologies offer a simple way to increase calculation speed by using multiple different computer devices, such as CPU and GPU. Using GPU as an additional computing unit can be used to greatly increase processing speed of CPU-bounded parallel tasks [10]. In-memory data storage is used to increase data access speed to stored data. A combination of those two technologies promises a significant performance increase for indexing algorithms. While B-trees are mostly limited by disk memory throughput and will greatly benefit from in-memory data storage, Aho-Corasick trees are more dependent on the processing speed and can be accelerated with use of GPGPU technology.

Lastly, as a further development of the described concept we also plan to develop and present versatile indexing algorithms for multiprocessor computer systems.

References

[1] A. A. Kozlov, A. A. Aleshina, I. S. Kamenskikh, M. M. Rovnyagin, D. M. Sinelnikov and D. A. Shulga, "Increasing the functionality of the modern NoSQL-systems with GPGPU-technology," 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), St. Petersburg, 2016, pp. 242-246, doi: 10.1109/EIConRusNW.2016.7448164.

[2] A. Lakshman, P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2 pp.35-40, 2010, doi: 10.1145/1773912.1773922.

[3] Hongjun Lu, Yuet Yeung Ng and Zengping Tian, "T-tree or B-tree: main memory database index structure revisited," Proceedings 11th Australasian Database Conference. ADC 2000 (Cat. No.PR00528), Canberra, ACT, Australia, 2000, pp. 65-73, doi: 10.1109/ADC.2000.819815.

[4] J. Ahn, C. Seo, R. Mayuram, R. Yaseen, J. Kim and S. Maeng, "ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys," in IEEE Transactions on Computers, vol. 65, no. 3, pp. 902-915, 1 March 2016, doi: 10.1109/TC.2015.2435779.

[5] D. Comer, "Ubiquitous B-tree", ACM Computing Surveys (CSUR), vol. 11, no. 2, pp. 121-137, 1979, doi: 10.1145/356770.356776.

[6] R. Sears, C. Van Ingen, J. Gray, "To blob or not to blob: Large object storage in a database or a filesystem?", arXiv preprint cs/0701168, 2007.

[7] S. Arudchutha, T. Nishanthy and R. G. Ragel, "String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm," 2013 IEEE 8th International Conference on Industrial and Information Systems, Peradeniya, 2013, pp. 231-236, doi: 10.1109/ICIInfS.2013.6731987.

[8] D. Herath, C. Lakmali and R. Ragel, "Accelerating string matching for bio-computing applications on multi-core CPUs," 2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS), Chennai, 2012, pp. 1-6, doi: 10.1109/ICIInfS.2012.6304784.

[9] M. M. Rovnyagin et al., "Modeling NoSQL Systems in Many-nodes Hybrid Environments," 2017 IEEE 11th International Conference on Application of Information and Communication Technologies (AICT), Moscow, Russia, 2017, pp. 1-4, doi: 10.1109/ICAICT.2017.8687028.

[10] I. S. Kamenskikh, D. M. Sinelnikov, D. S. Kalintsev, A. A. Kozlov, M. M. Rovnyagin and D. A. Shulga, "Software development framework for a distributed storage and GPGPU data processing infrastructure," 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), St. Petersburg, 2016, pp. 216-219, doi: 10.1109/EIConRusNW.2016.7448158.