УДК 004.04
DOI: 10.25559/SITITO.14.201801.111-125

A DYNAMIC INDEXING SCHEME FOR MULTIDIMENSIONAL DATA

Manuk G. Manukyan[1], Grigor R. Gevorgyan[2]

[1] Yerevan State University, Yerevan, Armenia

[2] Russian-Armenian (Slavonic) University, Yerevan, Armenia

Abstract

We present a new dynamic index structure for multidimensional data. The considered index structure is based on an extended grid file concept. Strengths and weaknesses of the grid files were analyzed. Based on that analysis we proposed to strengthen the concept of grid files by considering their stripes as linear hash tables, introducing the concept of chunk and representing the grid file structure as a graph. As a result we significantly reduced the amount of disk operations. Efficient algorithms for storage and access of index directory are proposed, in order to minimize memory usage and lookup operations complexities. Estimations of complexities for these algorithms are presented. A comparison of our approach to support effective grid file structure with other known approaches is presented. This comparison shows effectiveness of suggested metadata storage environment. An estimation of directory size is presented. A prototype to support of our grid file concept has been created and experimentally compared with MongoDB (a renowned NoSQL database). Comparison results show effectiveness of our approach in the cases of given point lookup, lookup by wide ranges and closest objects lookup when considering more than one dimension, and also better memory usage.

Keywords

Grid Files, Linear Hash Tables, Multidimensional Data, Data Warehouses.

ДИНАМИЧЕСКАЯ СХЕМА ИНДЕКСИРОВАНИЯ МНОГОМЕРНЫХ ДАННЫХ

М.Г. Манукян[1], Г.Р. Геворгян[2]

[1] Ереванский государственный университет, г. Ереван, Армения

[2] Российско-Армянский (Славянский) университет, г. Ереван, Армения

Аннотация

Мы представляем новую динамическую структуру индекса для многомерных данных.

About the authors:

Manuk G. Manukyan, Candidate of Physical and Mathematical Sciences, Associate Professor, Yerevan State University (1 Alex Manoogian St., Yerevan 0025, Republic of Armenia); ORCID: http://orcid.org/0000-0002-8578-3440, mgm@ysu.am

Grigor R. Gevorgyan, Ph.D. in Engineering Science, Russian-Armenian (Slavonic) University (123 Hovsep Emin St., Yerevan 0051, Republic of Armenia); ORCID: http://orcid.org/0000-0003-1706-568X, grigor.gevorgyan@gmail.com

© Manukyan M.G., Gevorgyan G.R., 2018

Рассматриваемая структура индекса основана на концепции сеточных файлов. Был проведен анализ сильных и слабых сторон сеточных файлов. На основе результатов данного анализа предложено усиление концепции сеточных файлов путем расмотрения полос (stripes) этих файлов как линейных хеш-таблиц, введения понятии сегмента (chunk) и представления структуры сеточного файла в виде графа. В результате мы существенно сократили количество дисковых операций. Предложены эффективные алгоритмы хранения и доступа к директории индекса, имеющие целью минимизацию затрат памяти и сложности операций поиска. Приведены оценки сложности этих алгоритмов. Предложено сравнение нашего подхода к поддержке эффективных сеточных файлов с известными подходами усиления концепций сеточных файлов. Сравнение показывает эффективность предложенной среды хранения для метаданных. Предложена оценка для размера директории. В нашем случае директория занимает минимальное количество байтов памяти. Разработан прототип для поддержки приведенной концепции сеточных файлов. Выполнено экспериментальное сравнение прототипа с MongoDB (широкоизвестная NoSQL база данных). Результаты сравнения показывают эффективность нашего подхода в случаях поиска по заданной точке, поиска по широким диапазонам значений и поиска ближайших соседних объектов в случаях использования более одного измерения, а также более эффективное использование памяти.

Ключевые слова

Сеточные файлы; линейные хеш-таблицы; многомерные данные; хранилища данных.

1. Introduction

The emergence of a new paradigm in science and various applications of information technology (IT) is related to issues of big data handling [18]. This concept involves the growing role of data in all areas of human activity beginning with research and ending with innovative developments in business. Such data is difficult to process and analyze using conventional database technologies. In this connection, the creation of new IT is expected in which data becomes dominant for new approaches to conceptualization, organization, and implementation of systems to solve problems that were previously considered extremely hard or, in some cases, impossible to solve. Unprecedented scale of development in the big data area and the U.S. and European programs related to big data underscore the importance of this trend in IT.

In the above discussed context we consider the concept of grid files [12, 15] as one of the adequate formalisms for effective management of big data. The grid file can be represented as if the space of points is partitioned in an imaginary grid. The *grid lines* parallel to axis of each dimension divide the space into *stripes*. The number of grid lines in different dimensions may vary, and there may be different distances between adjacent grid lines, even between lines in the same dimension. Intersections of these stripes form cells which hold references to data buckets containing records belonging to corresponding space partitions.

An example of 3-dimensional grid file is presented in Figure 1. Here X, Y and Z are the dimensions of considered 3D space, which is partitioned into stripes v_1, v_2, v_3 in X dimension, w_1, w_2 in Y dimension, and u_1, u_2, u_3 in Z dimension.

Dynamic aspects of file structures where all keys are treated symmetrically, avoiding distinction between primary and secondary keys, are studied in [12]. The article introduces the notions of a grid partition of the search space and of a grid directory, which are the keys to a dynamic file structure called the *grid file*. This file system is able to adapt to its contents under insertion and deletion operations, and thus achieves an upper bound of two disk accesses for single record retrieval. It also efficiently handles range queries and partially specified queries. Several splitting and merging policies, resulting in different refinements of the grid partition, are considered.

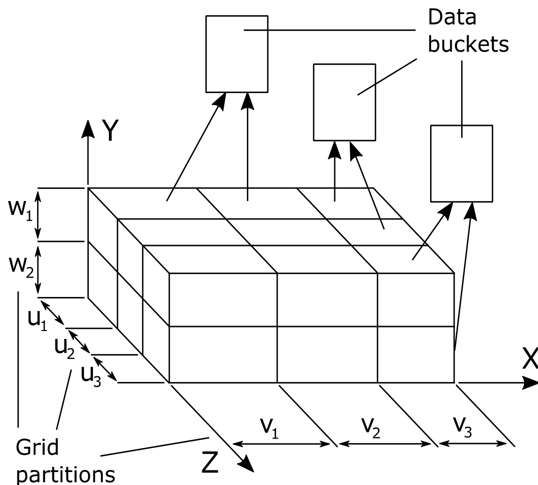


Figure 1. An example of 3-dimensional grid file

In [16] algorithms which generalize the standard lookup by single key techniques and apply them to search of records using several keys are specified. Two index file organization techniques, multidimensional dynamic hashing and multidimensional extendible hashing, which are multidimensional generalizations of dynamic and extendible hashing correspondingly, are specified and the average index size values for both cases, as well as their asymptotic expansions, are estimated. Multidimensional extensions of linear and extendible hash tables have also been proposed in [4, 13, 14].

In fact, the concept of grid files allows to effectively organize queries on multidimensional data [4] and can be used for efficient data cube storage in data warehouses [8, 15]. We have developed a data definition language of integrable data in the frame of a canonical data model based on the conception of grid files [9, 10, 11]. The weaknesses of grid file formalism are non-efficient memory usage by groups of cells referring to the same data buckets and the possibility of having a large number of overflow blocks for each data bucket [4].

In this paper an extension of the grid file concept is proposed addressing the above discussed weaknesses of grid file. First of all, we introduce the concept of the chunk: set of cells whose corresponding records are stored in the same data bucket (represented by single memory cells with one pointer to the corresponding data buckets). Chunking technique is used to solve the problem of empty cells in grid file. Second, we consider each stripe as a linear hash table which allows to increase the number of buckets more slowly (for each stripe, the average number of overflow blocks of chunks crossed by that stripe is less than one). By using this technique we essentially reduce number of disk operations. We also introduce an inner grid file structure representation as a directed acyclic graph, aiming to reduce the index directory volume. A directory definition language has been developed, which is independent from data management paradigms. Grid file operations are described and their complexities are estimated in the context of index modifications and disk operations. More detailed analysis of grid file operations and their algorithms have been suggested in [5, 6, 7].

This paper is organized as follows. An approach to grid file structure modification is proposed in Section 2. Further modification of this structure aiming to reduce the index file size is provided in Section 3. Section 4 contains representation of the grid file structure as a directed acyclic graph. Related work overview is set out in Section 5. Finally, conclusions are provided in Section 6.

2. Modification of the grid file structure

One of the problems intrinsic to grid files is the problem of non-efficient memory usage by groups of cells, referring to the same data buckets. We propose an alternative index structure, based on the grid file concept and aiming to avoid storage of duplicate pointers to the same data buckets, as well as to maintain slow index size growth and provide reasonable operation costs.

In this approach we do not store the grid file as a multidimensional array. The reason for this is that during each bucket split operation one of the stripes crossing it is also split into two stripes, thus doubling the number of cells of the original stripe, wherein many of the new cells contain duplicate pointers to the same data buckets. Instead, all cells whose corresponding records are stored in the same data buckets are grouped into *chunks*, represented by single memory cells with one pointer to the corresponding data buckets. Chunks are the main units for data input/output, as well as are used for data clusterisation. Chunks are used as a mechanism to solve the problem of empty cells in the grid file. For each dimension the information about its division is stored in a linear scale, each element of which corresponds to a stripe of the grid file and is represented as an array of pointers to the chunks, crossed by that stripe.

Each stripe is considered as a linear hash table. Overflow blocks are used to reduce the amount of grid file chunks. The number of overflow blocks may be different for different chunks, however we ensure that for any stripe the average number of overflow blocks for the chunks crossed by that stripe is less than one. This allows us to significantly reduce the total number of chunks, while guaranteeing not more than two disk operations for data access in average.

An example of 2-dimensional modified grid file is presented in Figure 2. It contains five chunks, each referencing to corresponding data bucket. Two chunks use overflow blocks. The solid lines represent borders between chunks, and dashed lines mean imaginary space divisions.

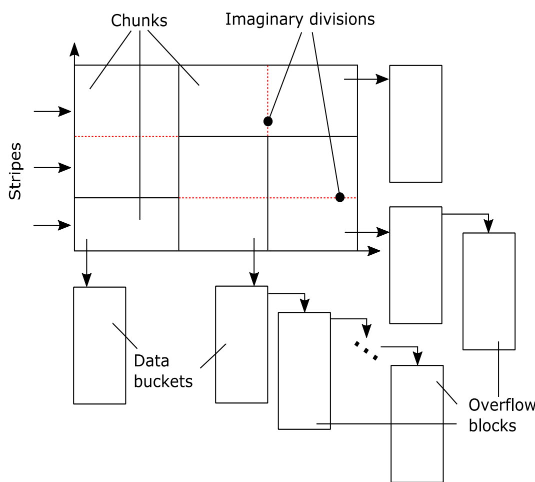


Figure 2. An example of 2-dimensional modified grid file

Splitting and merging policies similar to the ones described in [12] are used. Dimensions for both operations are selected so that the numbers of stripes in different dimensions do not differ by more than one. During the split operation we chose the dividing hyper plane in such a way that the numbers of points that end up in the resulting buckets differ as little as possible. For this purpose for each bucket we store statistical information about the average values of its points' coordinates. Merging is performed when the occupancy of resulting buckets is around 70 percent to achieve reasonable performance [12].

2.1. Estimations of the proposed concept characteristics

A grid file can be represented as a triple $F = \langle D, S, C \rangle$ where D is the set of dimensions, S is the set of stripes, and C is the set of chunks. Each stripe corresponds to exactly one dimension and crosses a non-empty subset of the set of chunks.

In this subsection we investigate several grid file characteristics and derive estimations of their values. Let us denote the number of dimensions as n and the average number of chunk split operations performed in one dimension as m.

Number of cells. Since each of n dimensions is divided into m parts in average, there exist m^n cells in average.

Number of stripes in one dimension. At most one new stripe is created during each split operation, and the number of stripes may be reduced during merge operations, so we can say that number of stripes in one dimension is limited by the number of split operations and has an order $O(m)$.

Total number of stripes has an order $O(nm)$ since there are $O(m)$ stripes in average in each of the n dimensions.

Total number of chunks. New chunks are created only during split operations, wherein as a result of one split operation the number of chunks is incremented. This means that the total number of chunks is limited by the total number of performed split operations and has an order $O(nm)$.

Number of cells per chunk in average is the ratio of total number of cells to the number of chunks and has an order $O\left(\frac{m^n}{nm}\right)$.

Average length of chunk side. The average length of a grid cell side, equal to the average stripe width, is used as the unit of measure. Without loss of generality assuming that the average shape of a chunk is an n-dimensional cube, the average length of its side has an order

$$O\left(\sqrt[n]{\frac{m^{n-1}}{n}}\right) = O\left(\frac{m}{\sqrt[n]{nm}}\right).$$

Average number of chunks crossed by a stripe. A stripe has an average length of m cells in n-1 dimensions. Since the average length of a chunk side is

$$O\left(\frac{m}{\sqrt[n]{nm}}\right),$$

a stripe will cross

$$O\left(\left(\frac{m}{\frac{m}{\sqrt[n]{nm}}}\right)^{n-1}\right) = O\left(\left(\sqrt[n]{nm}\right)^{n-1}\right)$$

chunks in average. For simplicity of reasoning we shall relax this estimation to $O(nm)$.

Directory size. Since each of the $O(nm)$stripes crosses $O(nm)$chunks in average, the total number of stored references will be $O(n^2m^2)$. Also each

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

chunk stores one reference to the corresponding data bucket. Hereby, the grid file directory size is $O(n^2m^2)$.

2.2. Grid file operations

In this section we consider grid file operations, adapted for use with the proposed index structure. Operation cost estimations are provided in the context of index modification and disk operations.

Lookup by key. In order to find all records for a given value d_0 of dimension d we first find the stripe s of that dimension which d_0 belongs to, then consider all the chunks crossed by s. For each of these chunks it is necessary to load corresponding data blocks and check if they contain records matching the query. Since all of these chunks have to be considered, and no other chunk may contain any query results, only necessary and sufficient chunks will be considered. The average number of such chunks is $O(nm)$ according to Section 2.1. The stripe corresponding to the given coordinate can be found in $O(\log m)$ time using binary search on the sorted linear scale of the given dimension. Hereby the complexity of lookup by key operation is $O(nm \log m)$. Besides that, it is possible to use effective data structures for stripe lookup, such as van Emde Boas trees [1]. In [5, 6] a method of van Emde Boas trees usage when points' coordinates are represented as 32 bit integers is described, allowing to significantly reduce the amount of operations for stripes lookup compared to logarithmic algorithms. Number of disk operations, due to overflow blocks usage, in average equals to doubled amount of considered chunks and has an order $O(nm)$. In [5, 6] is described also a method of hash-functions usage to exclude chunks which a priori do not contain any records matching the query from consideration, which allows to reduce the number of disk operations.

Lookup by several coordinates. To find values by given k coordinates we first find the stripes, corresponding to them, then intersect the sets of chunks crossed by these stripes and consider only the data buckets, corresponding to the chunks of intersection. Similarly to the conclusions provided in the description of lookup by key operation, the stripes lookup requires $O(k \log m)$ time. To find the intersection of chunk pointers sets of obtained stripes, we consider chunks crossed by one of these stripes, and for each of them check if it is crossed by other stripes as well. This takes O(k) time, since we can check if a stripe crosses a chunk by comparing coordinates of their bounds. Hereby, the sets intersection operation requires $O(knm)$ time, and the total complexity of lookup by k coordinates operation is $O(k \log m + knm)=O(knm)$.

To perform intersection of sets of chunks crossed by obtained stripes fast set intersection algorithms [3] can be used, allowing to achieve effective bound of $O\left(\frac{knm}{\sqrt{w}}\right)$, where w is the size of machine word. It should be noted that further modification of the grid file structure, proposed in Section 3, makes it difficult to apply such algorithms.

Given point lookup is a special case of the previous operation where $k=n$. The complexity of this operation is $O(n^2m)$, and the usage of fast set intersection algorithms allows to reduce it to $O\left(\frac{n^2m}{\sqrt{w}}\right)$. Since in this case only one chunk, containing the given point, has be considered, the amount of disk operations in average does not exceed 2.

Closest objects lookup. To find the points, closest to the provided one, we first find the chunk it belongs to, and check the corresponding data buckets. It may happen that, however, the closest point belongs to one of the adjacent chunks and they need to be considered as well. In the worst case we shall need to consider $2n$ adjacent chunks. Hereby, the complexity of closest objects lookup operation does not differ from the complexity of point lookup operation. However, the amount of disk operations can increase when considering adjacent chunks -- in the worst case $2(2n+1)$ disk operations will be required.

Lookup by ranges of values. This operation is similar to the lookup by several coordinates operation, but in this case in each dimension we may need to consider more than one stripe. This makes it difficult to use fast set intersection algorithms, since the sets of chunks, crossed by stripes of the same dimension, have to be united first. Assuming that given ranges will cover t stripes in each dimension in average, similarly to the estimation of lookup by several coordinates operation we get that complexity of this operation is $O(tnmk)$. The amount of disk operations in average equals to the doubled number of considered chunks.

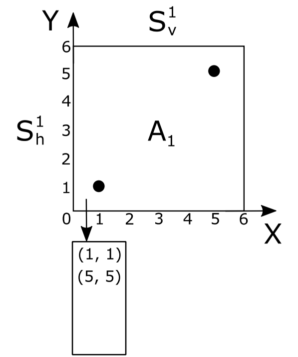
Insert. To insert a value we first locate the chunk it belongs to by given coordinates, similarly to given point lookup operation, then load the corresponding data block and perform insertion. Since we consider each stripe as a linear hash table, in case of insufficient space it may be possible to

Том 14, № 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

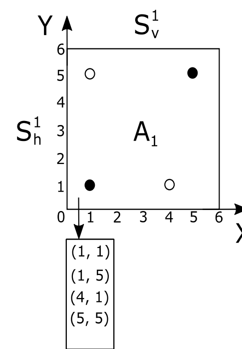
add an overflow block. However, if after insertion there exists a stripe for which the average number of overflow blocks for the chunks crossed by that stripe exceeds 1, we need to split current chunk into two parts and reorganize corresponding data buckets.

Split. Consider a chunk *c* which is necessary to split. First, we choose the dimension in which the split shall be performed. For that purpose we choose such dimension *d* which currently contains the least number of stripes. This is done to maintain symmetry of the grid file structure regarding its dimensions. Then we choose some coordinate d_0 in dimension *d* which defines the dividing hyperplane. Chunk *c* is then split into two chunks c_1 and c_2, and all records from segment *c* for which the corresponding points have coordinate in dimension *d* less than d_0 are stored in c_1, and the rest stored in c_2. The coordinate d_0 is chosen in such a way that the numbers of points in the resulting chunks are as close as possible. As a result of split operation, some stripe *s* in dimension *d* is also divided into two stripes s_1 and s_2. Pointers to the newly created chunks are added to pointer lists of all stripes crossing them. According to Section 2.1, the sets of chunk pointers for the newly created stripes consist of *O(nm)* elements. Also *O(nm)* pointers to the new chunks are added to the sets of pointers of other dimensions stripes. Hereby the split operation complexity is *O(nm).* To redistribute the records we shall need in average not more than 6 disk operations - 2 to read the data of chunk *c* and 4 to write data of chunks c_1 and c_2.

As an example, let us consider a 2-dimensional grid file with coordinate axis *X* and *Y*. Assume that each data block has capacity enough to contain not more than 5 records. Analogously to linear hash tables, we shall ensure that in each stripe the ratio of average number of records in a block to block size does not exceed some constant value [4]. We shall use the value of 80\% for this purpose. Let us denote number of records in a stripe as *r* and number of chunks in a stripe as *c*. From the above condition we get that r/c\leq4 inequality must take place. Let us also denote *r/c* ratio as *k*.



$S_h^1=\{A_1\}$ $S_v^1=\{A_1\}$



$S_h^1=\{A_1\}$ $S_v^1=\{A_1\}$

Initial state (b) Inserted (1, 5) and (4, 1)

Figure 3. A simple insert operation

At first, we have one chunk A_1, containing two points *(1, 1)* and *(5, 5*). We also have one horizontal stripe S_h^1 and one vertical stripe S_v^1. Then we insert two points *(1, 5)* and *(4, 1).* After that, for each stripe *r=4, c=1* and *k=4*, so we can proceed without splitting the chunk. Since there is enough space in data bucket, the records will be just added there. This process is illustrated in Figure 3.

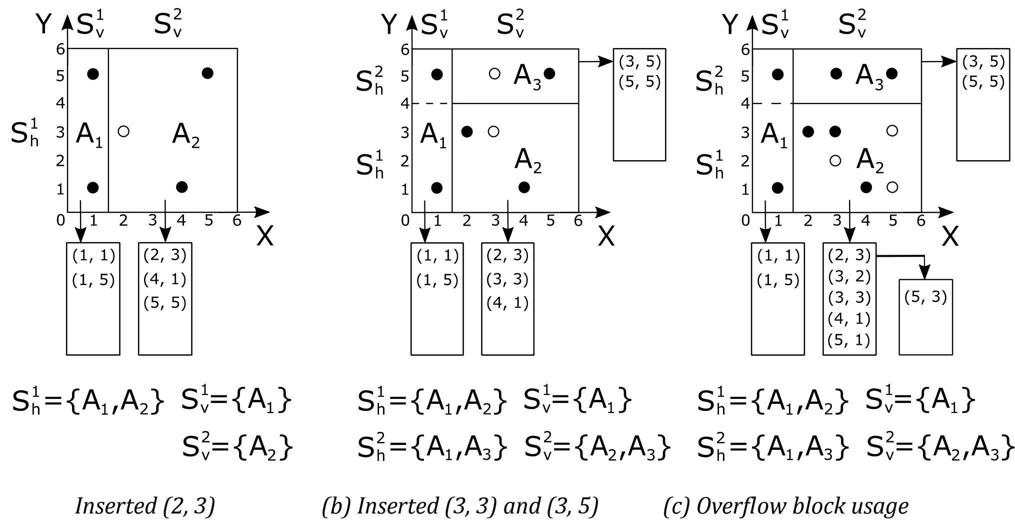
Let us insert another point *(2, 3).* If we insert it into chunk A_1 we shall get *r=5, c=1* and *k=5>4*, so we have to split the chunk. After splitting it vertically with *x=1,5* line, we obtain two chunks A_1 and A_2, as well as another vertical stripe S_v^2. Now for stripe S_h^1 parameter *c* equals to *2*, because S_h^1 now consists of two chunks, and *k=2,5* allows us to proceed with insertion.

It is fine to insert another point *(3, 3)* into chunk A_2, but after adding one more point *(3, 5)* we face a necessity to split the chunk A_2, because this time *k=5>4* for stripe S_v^2. We split it horizontally with line *y=4*. That line also crosses chunk A_1, but we do not split it too. The result of above operations is

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

illustrated in Figure 4 (a) and (b). We represent the imaginary split of chunk A_1 with a dashed line.

Finally, let us insert 3 more points: *(3, 2)*, *(5, 1)* and *(5, 3)*. There is enough space for *(3, 2)* and *(5, 1)* in chunk A_2, however the third point *(5, 3)* does not fit there because the corresponding bucket is already full. However, for both stripes S_h^1 and S_v^2 crossing the chunk A_2 we get *r=8*, *c=2* and *k=4*, so we can avoid splitting the chunk (and increasing directory size) by using an overflow block. The result of such insertion is presented in Figure 4 (c).



Inserted (2, 3) *(b) Inserted (3, 3) and (3, 5)* *(c) Overflow block usage*

Figure 4. Split operations

A single split operation increments the number of chunks and the number of stripes. Per Section 2.1, the new stripe pointer list will contain *O(nm)* elements in average. Also, *O(nm)* pointers will be added into pointer lists of all other dimensions. Hence, split operation will increase directory size by *O(nm).*

Delete. To delete a value, as when inserting, we first find the chunk it belongs to and read the corresponding data block. After deletion the considered data bucket may become empty. In such case we can merge this chunk with another adjacent chunk. It is possible to merge chunks even if the data bucket is not empty, if in the result for each stripe the condition of having not more than one overflow block for the chunks crossed by that stripe is satisfied. Complexity of delete operation equals to the complexity of insert operation.

Merge. Consider two chunks c_1 and c_2 which have a common boundary in dimension *d* and are being merged into a single chunk *c*. For any stripe *s* if its pointer list contains either a pointer to c_1 or c_2, those should be replaced with a pointer to *c* instead. Complexity of merge operation equals to the complexity of split operation. A single merge operation will remove one chunk and the number of pointers equal to the number of stripes crossing the merged chunks - *O(nm)* in average.

3. Alternative grid file structure

In this section an alternative grid file structure is proposed, which is a further modification of the structure, proposed in Section 2, and allows to reduce directory size from $O(n^2m^2)$to $O(n^2m)$. To achieve this goal we reorganize the pointers storage structure, allowing chunks to store pointers to each other.

Definition 1. Let there be a total order < defined on the set of chunks. Let us also denote the projection of chunk *c* to dimension *d* as $\pi_d(c)$

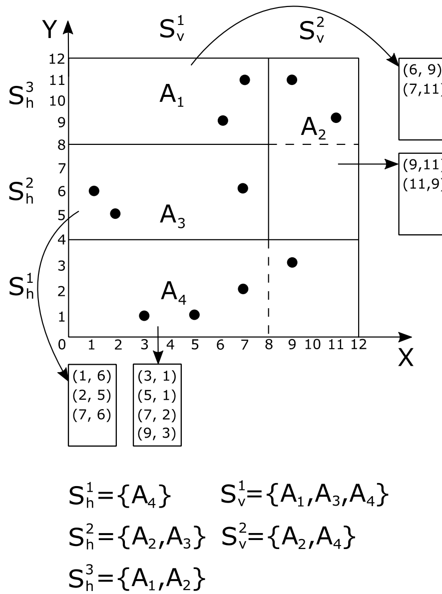
The set of pointers *R* is defined as follows:

1. For each pair of chunks *a* and *b*, s.t. *a* < *b* and a dimension *d* exists s.t. $\pi_d(a)\pi_d(b)$, and no chunk *c* exists s.t. *a* < *c* < *b* and $\pi_d(a)\pi_d(c)\pi_d(b)$, there exists a pointer $(a, b) \in R$. Let us call such pointer a *pointer in dimension d*;

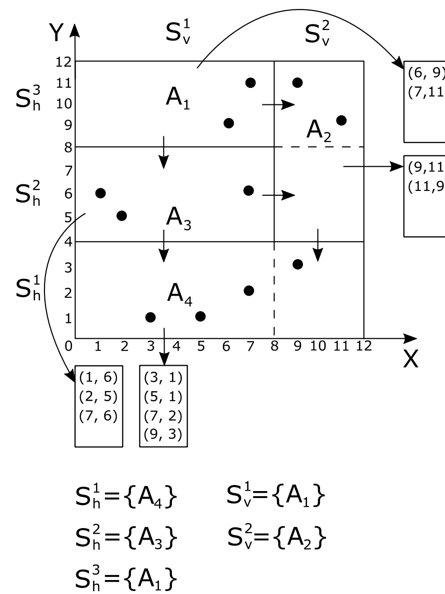
2. For each chunk *a* and stripe *s* of dimension *d*, if in *R* there exists no pointer to segment *a* in dimension *d*, then there exists a pointer $(s, a) \in R$.

An example of a grid file constructed according to initial structure and its modification are presented in Figure 5. Note that in this case in

modified structure each stripe's pointer list contains a pointer to only one chunk, and the chunks contain p ointers to each other according to Definition 1.



(a) Grid file constructed according to initial structure



(b) Modified grid file representation

Figure 5. Grid file structure modification

It has been proved in [5, 6] that the expected directory size to store this structure is $O(n^2m)$.

3.1. Comparison of directory size

This section compares the obtained directory size with two techniques for grid file organization proposed in [16]

- multidimensional dynamic hashing (MDH) and multidimensional extendible hashing (MEH). Directory sizes for both of these techniques are also provided in [16]: $O\left(r^{1+\frac{1}{s}}\right)$ for MDH and $O\left(r^{1+\frac{n-1}{ns-1}}\right)$for MEH, where r is the total number of records, s is the block size and n is the number of dimensions. It should be noted that we consider the case of uniform distributions.

For comparison let us express the directory size in case of our approach using these values. Since we allow each chunk to have one overflow block in average, we can without loss of generality assume that each of the overflow blocks will be half-full in average, meaning that we shall store $\frac{3}{2}s$ records per chunk in average. Hereby we can conclude that it will be required to have $\frac{2r}{3s}$chunks to store all r records, which is equivalent to *O(nm)* according to Section 2.1. Hereby, according to Section 3, our directory size can be estimated as $O(n^2m) = O\left(\frac{nr}{s}\right)$.

Compared to MDH and MEH techniques, directory size in our approach is $\frac{sr^{\frac{1}{s}}}{n}$ and $\frac{sr^{\frac{n-1}{ns-1}}}{n}$ times smaller correspondingly.

3.2. Grid file operations

Lookup, insert and delete operations. A significant difference from the structure proposed in Section 2 is that the sets of chunks crossed by a stripe are not stored separately for each stripe. To visit all chunks crossed by stripe s of dimension d, we have to consider all chunks, pointers to which are stored in the pointer list of stripe s, and for each of them iterate by the pointers of dimension d. This makes it difficult to use fast set intersection algorithms in this case, since the required sets cannot be presented in the form of necessary structures in advance.

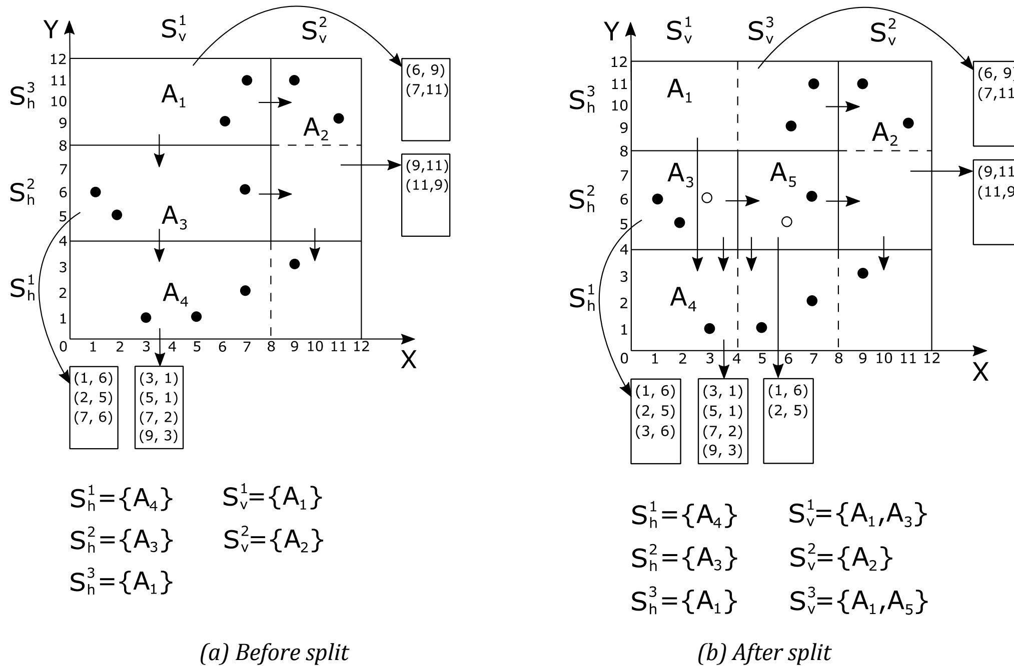
Merge and split operations. It has been shown in [9, 10] that the time complexity of merge and split operations is $O(n)$.

As an example, consider splitting chunk A_3 of grid file represented in Figure 5(b) after insertion of points *(3, 6)* and *(6, 5)*. A new stripe S_v^3 and chunk A_5 are created and integrated into the structure of the grid file. The result of such operation is

illustrated in Figure 6.

4. Grid file as a directed acyclic graph

The structure of our model allows us to naturally represent the grid file as a directed acyclic graph. Such representation is used as an implementation of our index. Hereby, we came to a method of effective representation of an index structure, based on the concept of grid file, by means of graph theory.



(a) Before split *(b) After split*

Figure 6. Vertical split of chunk A_3

Definition 2. We say that the graph $G = \langle V, E \rangle$ represents the grid file $F = \langle D, S, C \rangle$, if it is formed according to the following rules:

- For each chunk $a \in C$ there exists a vertex $v_a \in V$
- For each stripe $s \in S$ there exists a vertex $v_s \in V$
- For each pointer $(a, b) \in R$, where a is a stripe or a segment and b is a segment, there exists an oriented edge $(v_a, v_b) \in E$

Figure 7 illustrates the graph representation of the grid file from Figure 6(b).

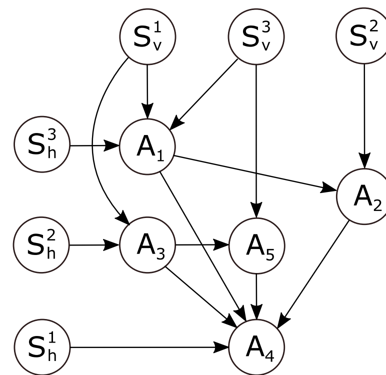


Figure 7. Grid file graph representation

5. Related work

There is a variety of approaches for big data management today. We shall review some of them, briefly describe their data models, approaches for scaling and pay attention to their data indexation methods. More detailed analysis of big data systems can be found in [18].

SciDB [19, 20] is an open-source data management system intended primarily for use in application domains that involve very large (petabyte) scale array data. SciDB uses array data model, and its databases are organized as collections of n-dimensional arrays. Data stored in the arrays is being vertically partitioned. The SciDB storage manager splits attributes in a single logical array and handles values for each attribute separately. Vertical partitioning, similarly to column-storage system, reduces I/O costs. Also SciDB storage manager takes each attributes data, and further decomposes the array into a number of equal sized, and potentially overlapping, chunks. In SciDB chunks are the physical units of I/O, processing, and inter-node communication. SciDB uses array indexing. Coordinates are used for determination of the chunk, containing requested data. Non-integer dimensions are supported by indexes, mapping dimension values into integers.

Couchbase [http://www.couchbase.com] is an open-source document-oriented distributed NoSQL database for interactive applications, designed for easy scalability, consistent high-performance and 24/7 availability. Couchbase Server stores data as JSON format documents or binary data. Documents are hashed by IDs and then uniformly distributed across the cluster and stored in data containers called buckets, which represent logical groupings of physical resources within a cluster.

Documents stored at Couchbase Server can be indexed. Secondary indexes are defined using design documents and views. Each design document can have multiple views, and each Couchbase bucket can have multiple design documents. Couchbase indexes are distributed, with each server indexing only the data it contains.

Column-oriented databases originate from Google's **Big Table** [2]. Its data model is based on a sparsely populated table whose rows can contain arbitrary columns, the keys for which provide natural indexing. Each cell in BigTable can contain multiple versions of the same data, indexed by timestamp. Different variations of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

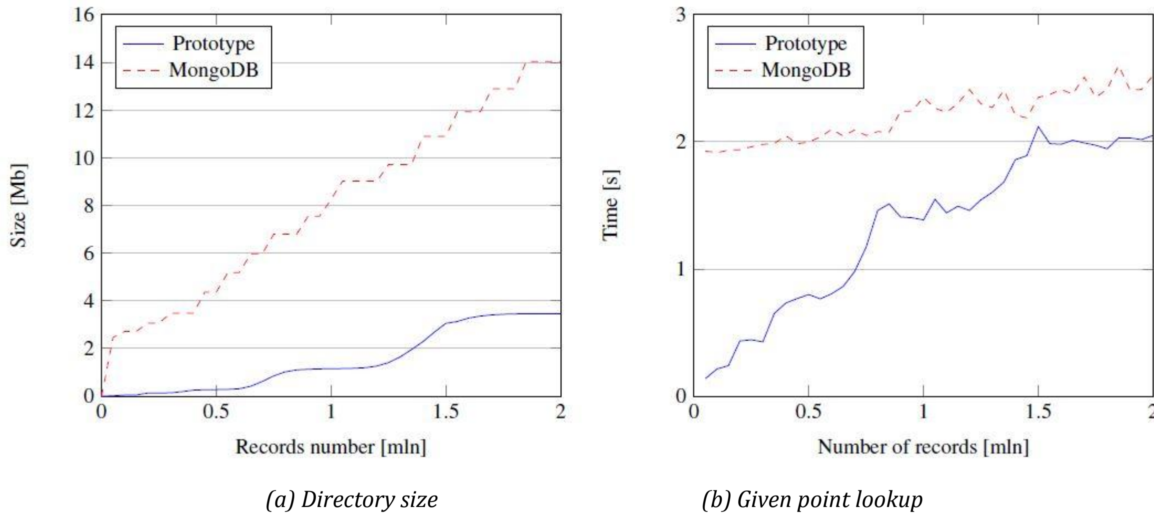
Apache Cassandra [http://cassandra.apache.com] is designed to handle big data workloads across multiple nodes with no single point of failure. Nodes are the basic infrastructure components in Cassandra. Data is distributed among all nodes in the cluster, indexed and written to an in-memory structure, called a memtable. Once it is full, the data is written to disk in an SSTable (Sorted Strings Table) data format. The SSTable concepts is borrowed from Google's BigTable, it stores data as a set of row fragments in sorted order, based on row keys. The index structure of SSTable is a sparse index, which maps row keys to corresponding offsets in the data file.

Neo4j [http://neo4j.com, 17] is a representative of graph databases. It stores data as a graph, where it is represented as nodes and relationships, which can have properties. Besides relationships, nodes can also have several labels. Querying in neo4j is performed using Cypher - a declarative, SQL-inspired language for describing patterns in graphs. Indexes are created per label and property combinations.

MongoDB [https://www.mongodb.org] is an open-source document-oriented No-SQL database, providing high performance, availability and easy scalability. The database consists of collections, representing sets of MongoDB documents. Each document is a set of key-value pairs, and is stored in BSON (binary serialized JSON) format. MongoDB supports different types of indexes, namely: single field, compound, multikey, geospatial, text and hashed indexes. MongoDB indexes are implemented as B-Trees.

5.1. Comparison with MongoDB

We have implemented a data warehouse prototype based on the proposed dynamic indexation scheme and compared its performance with MongoDB. MongoDB was chosen for comparison for pragmatical reasons since it is currently one of the most demanded NoSQL databases. Testing was conducted using four main query categories [4] - given point lookup, lookup by individual coordinates, range lookup and closest object lookup. Detailed testing results and description of several techniques used for prototype implementations are listed below.



(a) Directory size *(b) Given point lookup*

Figure 8. Directory size and given point lookup

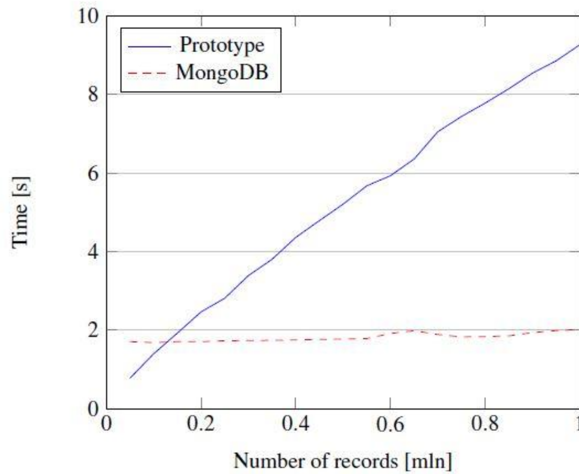
Directory size. Figure 8(a) presents charts of index directory size growth when performing 2 mln. insert operations. It can be seen that our data warehouse prototype index structure requires much less memory than B-trees used by MongoDB.

Given point lookup. Given all the coordinates, as described in Section 2.2, we first find the corresponding stripes in each dimension, then intersect the sets of chunks crossed by those stripes. That intersection consists of the only chunk, containing the necessary point. Van Emde Boas trees are used for efficient stripe lookup. The considered prototype implementation uses 32-bit integers to represent values in each dimension. Stripes are grouped in accordance with 16 most significant bits of their lower bounds. Usage of van Emde Boas trees allows to find the necessary group in $log\ log\ 2^{16} = 4$ operations. Then a binary search is performed within the obtained group. Since only lower 16 bits are significant within a group, it takes total of $log\ 2^{16} + 4 = 20$ (at most) operations for stripe lookup. Fast set intersection algorithms are also used, as described in Section 3.3, allowing to effectively find the necessary chunk.

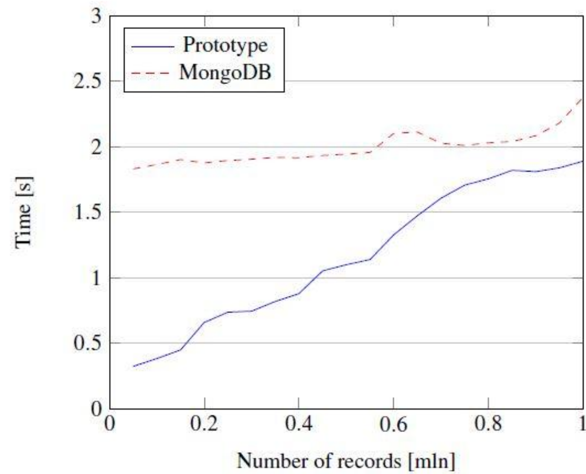
Figure 8(b) presents charts, displaying the time required to process 10^5 given point lookup operations depending on the number of records in database. It can be seen that our data warehouse prototype index structure requires much less memory than B-trees used by MongoDB and processes point lookup queries faster. It can be seen that our prototype performs nearly 3 times faster in this case. Also let us note that the tendency of this difference is independent from the number of records in the database and remains the same with its increase. This note applies to all further test cases as well.

Lookup by individual coordinates. This algorithm is similar to the previous one. The difference is that not all the dimensions may be present in query, and the resulting chunk set intersection may consist of numerous chunks which have to be considered. However, many of these chunks may not actually contain any records matching the query. To avoid unnecessary disk read operations for such chunks, we try to identify them using a method similar to used in [3] for fast set intersection. We use several independent hash functions $h_1 \ldots h_k$ which map each coordinate value into the range $[1..w]$, where w is the size of machine word. For each chunk c_i and dimension d we store words w_{ij}^d $(j=1..k)$, and when a value with coordinate v^d is inserted in chunk c_i we calculate values $h_1(v^d),\ldots,h_k(v^d)$ and set corresponding bits in w_{ij}^d. During lookup we calculate values of hash functions in the same manner, and check the corresponding bits - if they are not set, we can omit loading the considered chunk.

Figure 9 illustrates performance of our prototype compared to MongoDB when performing lookup by 1 and 2 dimensions on the same database as described in previous paragraph. It can be seen that our approach manifests itself better when more dimensions are present in the query.



(a) Lookup by 1 dimension

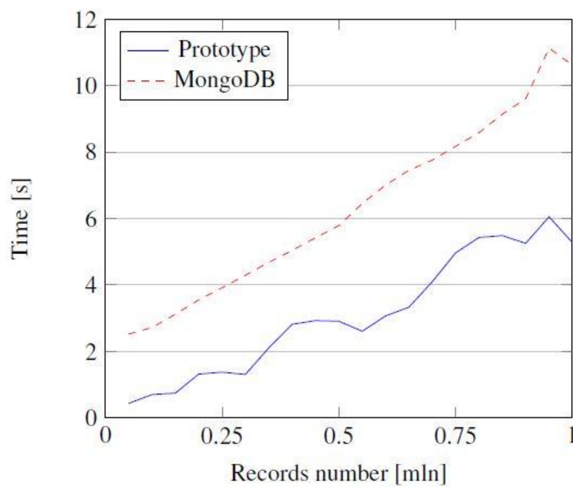


(b) Lookup by 2 dimensions

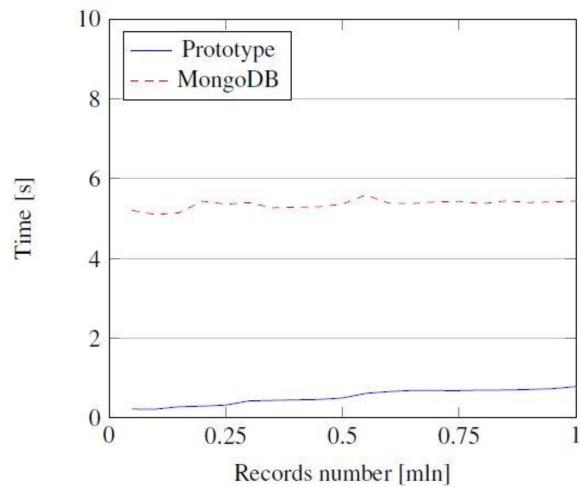
Figure 9. Lookup by individual coordinates

Range lookup. A query for data range lookup defined a rectangular area of the grid, and answer to it is composed by set of points belonging to the chunks which are crossed by that area. Figure 10(a) contains comparison of range queries performed in 3-dimensional space with values uniformly distributed in range [*$0..2^{32}$-1*] and average range length of *10^6*. It should be noted that with increasing of query range length our prototype performs better in comparison with MongoDB.

Closest object lookup. To compare performance on closest point lookup queries we used a database of 10^6 2-dimensional points to make use of geospatial indexes of MongoDB. Comparison results are illustrated in Figure 10 (b). It should be noted that our approach can handle closest object lookup queries in higher dimensional cases as well, while MongoDB provides geospatial indexes only for 2-dimensional points.



(a) Range lookup



(b) Closest object lookup

Figure 10. Range and closest object lookup

6. Conclusions

In the present paper a new dynamic indexing scheme for effective management of big data is proposed. Our approach to such index structure is based on the grid file concept. In introduction

strengths and weaknesses of grid file concept are discussed. Based on this analysis the grid file concept has been extended in order to eliminate the considered weaknesses of grid file. Namely, we introduce the concepts of chunk, used to solve the problem of empty cells in grid file, and modify the concept of stripe, essentially restricting the number of disk operations.

Estimates of the characteristics of our concept of grid file were obtained. In particular, for directory size we received the following estimation: $O(n^2 m^2)$, where n - the number of dimensions and m - the average number of splits performed in one dimension. Such estimation is the result of our chunk concept introduction.

Further improvement of the index structure leads us to the following estimation of directory size: $O(n^2 m)$. This result is very important since m is a parameter. The improved index structure allows to effectively reduce memory usage and complexities of split and merge operations. Efficient algorithms for storage and access of grid file directory are proposed in order to minimize memory usage and lookup operations complexities. Estimations of complexities for these algorithms are presented.

We performed comparison of our approach directory size with two techniques (MDH, MEH) for grid file organization proposed in [16]. Compared to MDH and MEH techniques, directory size in our approach is $\frac{sr^{\frac{1}{s}}}{n}$ and $\frac{sr^{\frac{n-1}{ns-1}}}{n}$ times smaller correspondingly, where r is the total number of records, s is the block size and n is the number of dimensions.

Finally, results of experimental comparison of a prototype, implementing the proposed concept of grid file, with MongoDB (one of the most demanded NoSQL databases) show that our prototype is effective in the cases of point lookup, lookup in wide ranges, lookup of closest objects and also has more effective memory usage.

REFERENCES

[1] Boas P. Van Emde, Kaas R., Zijlstra E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*. 1976; 10(1):99-127. DOI: https://doi.org/10.1007/BF01683268

[2] Chang F., Dean J., Ghemawat S., Hsieh W.C., Wallach D.A., Burrows M., Chandra T., Fikes A., Gruber R.E. Bigtable: a distributed storage system for structured data. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, Washington, November 06 – 08, 2006. p. 205-218. Available at: http://tab.d-thinker.org/showthread.php?tid=4 (accessed 11.01.18).

[3] Ding B., Konig A.C. Fast Set Intersection in Memory. *Proceedings of the 37th International Conference on Very Large Databases (VLDB 2011)*, Seattle, Washington, USA: Very Large Data Bases Endowment Inc., 2011; 4(1-12):255–266. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/p255-DingKoenig.pdf (accessed 11.01.18).

[4] Garcia-Molina H., Ullman J., Widom J. Database Systems: The Complete Book. Prentice Hall, USA. 2009. Available at: http://infolab.stanford.edu/~ullman/dscb.html (accessed 11.01.18).

[5] Gevorgyan G. An Effective Dynamic Structure for Grid file Organization. *Russian-Armenian (Slavonic) University Bulletin*. 2016; (l):5-17. Available at: https://elibrary.ru/item.asp?id=30752424 (accessed 11.01.18).

[6] Gevorgyan G. An Approach to Data Integration: Model, Algorithms and Verification // PhD thesis: National Academy of Sciences of the Republic of Armenia, Yerevan, 2016.

[7] Gevorgyan, G., Manukyan, M. Effective Algorithms to Support Grid Files. *Russian-Armenian (Slavonic) University Bulletin*. 2015; (2):22–38. Available at: https://elibrary.ru/item.asp?id=30752417 (accessed 11.01.18).

[8] Luo C., Hou W.C., Wang C.F., Want H., Yu X. Grid File for Efficient Data Cube Storage. Computers and their Applications. 2006. p. 424–429.

[9] Manukyan M., Gevorgyan G. Canonical Data Model for Data Warehouse. *Communications in Computer and Information Science*. 2016; 637:72-79. DOI: https://doi.org/10.1007/978-3-319-44066-8_8

[10] Manukyan M. Canonical Model: Construction Principles. *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS '14)*, Maria Indrawan-Santiago, Matthias Steinbauer, Hong-Quang Nguyen, A. Min Tjoa, Ismail Khalil, and Gabriele Anderst-Kotsis (Eds.). ACM, New York, NY, USA, 2014. p. 320-329. DOI: http://dx.doi.org/10.1145/2684200.2684278

[11] Manukyan M. On an Approach to Data Integration: Concept, Formal Foundations and Data Model. *CEUR Workshop Proceedings*. 2017; 2022:206-213. Available at: http://ceur-ws.org/Vol-2022/paper34.pdf (accessed 11.01.18).

[12] Nievergelt J., Hinterberger H., Sevcik K. The Grid File: An Adaptable, Symmetric, Multikey File Structure. *ACM Transactions on Database Systems*. 1984; 9(1):38–71. DOI: http://dx.doi.org/10.1145/348.318586

[13] Otoo E.J. A Mapping Function for the Directory of a Multidimensional Extendible Hashing. *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB '84)*, Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1984. p. 493-506.

[14] Otoo E.J. A multidimensional digital hashing scheme for files with composite keys. *Proceedings of the 1985 ACM SIGMOD international conference on Management of data (SIGMOD '85)*. ACM, New York, NY, USA, 1985. p. 214-229. DOI: http://dx.doi.org/10.1145/318898.318918

[15] Papadopoulos A.N., Manolopoulos Y., Theodoridis Y., Tsoras V. Grid File (and family). Encyclopedia of Database Systems, Springer, Boston, MA, 2009. p. 1279–1282.
[16] Regnier M. Analysis of Grid File Algorithms. *BIT - Computer Science Section*. 1985; 25(2):335–358.
[17] Robinson I., Webber J., Eifrem E. Graph Databases. 2nd ed. O'Reilly, USA, 2015. 237 p.
[18] Sharma S., Tim U.S., Wong J., Gadia S., Sharma S. A Brief Review on Leading Big Data Models. *Data Science Journal*. 2014; 13:138-157. DOI: https://doi.org/10.2481/dsj.14-041
[19] Stonebraker M., Brown P., Poliakov A., Raman S. The Architecture of SciDB. In: Bayard Cushing J., French J., Bowers S. (eds) Scientific and Statistical Database Management. SSDBM 2011. Lecture Notes in Computer Science. Vol. 6809. Springer, Berlin, Heidelberg, 2011. DOI: https://doi.org/10.1007/978-3-642-22351-8_1
[20] Brown P.G. Overview of SciDB: large scale array storage, processing and analysis. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10)*. ACM, New York, NY, USA, 2010. p. 963-968. DOI: https://doi.org/10.1145/1807167.1807271

Submitted 11.01.2018; Revised 10.02.2018; Published 30.03.2018.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] *Boas P. Van Emde, Kaas R., Zijlstra E.* Design and implementation of an efficient priority queue // Mathematical Systems Theory. 1976. Vol. 10, issue 1. Pp. 99-127. DOI: https://doi.org/10.1007/BF01683268
[2] Bigtable: a distributed storage system for structured data / F. Chang [et al.] // Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, Washington, November 06 – 08, 2006. Pp. 205-218. URL: http://tab.d-thinker.org/showthread.php?tid=4 (дата обращения: 11.01.18).
[3] Fast Set Intersection in Memory / B. Ding, A.C. Konig // Proceedings of the 37th International Conference on Very Large Databases (VLDB 2011), Seattle, Washington, USA: Very Large Data Bases Endowment Inc., 2011. Vol. 4, no. 1-12. Pp. 255–266. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/p255-DingKoenig.pdf (дата обращения: 11.01.18).
[4] *Garcia-Molina H., Ullman J., Widom J.* Database Systems: The Complete Book. Prentice Hall, USA. 2009. URL: http://infolab.stanford.edu/~ullman/dscb.html (дата обращения: 11.01.18).
[5] *Геворгян Г.* Эффективная динамическая структура для организации сеточных файлов // Вестник Российско-Армянского (Славянского) университета: Физико-математические и естественные науки. 2016. № 1. С. 5-17. URL: https://elibrary.ru/item.asp?id=30752424 (дата обращения: 11.01.18).
[6] *Gevorgyan G.* An Approach to Data Integration: Model, Algorithms and Verification // PhD thesis: National Academy of Sciences of the Republic of Armenia, Yerevan, 2016.
[7] *Геворгян Г., Манукян М.* Эффективные алгоритмы для поддержки сеточных файлов // Вестник Российско-Армянского (Славянского) университета: Физико-математические и естественные науки. 2015. № 2. С. 22–38. URL: https://elibrary.ru/item.asp?id=30752417 (дата обращения: 11.01.18).
[8] Luo C., Hou W.C., Wang C.F., Want H., Yu X. Grid File for Efficient Data Cube Storage // Computers and their Applications. 2006. Pp. 424–429.
[9] *Manukyan M., Gevorgyan G.* Canonical Data Model for Data Warehouse. Communications in Computer and Information Science. 2016. Vol. 637. Pp. 72-79. DOI: https://doi.org/10.1007/978-3-319-44066-8_8
[10] Canonical Model: Construction Principles / M. Manukyan // Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS '14), Maria Indrawan-Santiago, Matthias Steinbauer, Hong-Quang Nguyen, A. Min Tjoa, Ismail Khalil, and Gabriele Anderst-Kotsis (Eds.). ACM, New York, NY, USA, 2014. Pp. 320-329. DOI: http://dx.doi.org/10.1145/2684200.2684278
[11] *Manukyan M.* On an Approach to Data Integration: Concept, Formal Foundations and Data Model // CEUR Workshop Proceedings. 2017. Vol. 2022. Pp. 206-213. URL: http://ceur-ws.org/Vol-2022/paper34.pdf (дата обращения: 11.01.18).
[12] *Nievergelt J., Hinterberger H., Sevcik K.* The Grid File: An Adaptable, Symmetric, Multikey File Structure // ACM Transactions on Database Systems. 1984. Vol. 9, issue 1. Pp. 38–71. DOI: http://dx.doi.org/10.1145/348.318586
[13] A Mapping Function for the Directory of a Multidimensional Extendible Hashing / E.J. Otoo // Proceedings of the 10th International Conference on Very Large Data Bases (VLDB '84), Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1984. Pp. 493-506.
[14] A multidimensional digital hashing scheme for files with composite keys / E.J. Otoo // Proceedings of the 1985 ACM SIGMOD international conference on Management of data (SIGMOD '85). ACM, New York, NY, USA, 1985. Pp. 214-229. DOI: http://dx.doi.org/10.1145/318898.318918
[15] *Papadopoulos A.N., Manolopoulos Y., Theodoridis Y., Tsoras V.* Grid File (and family). Encyclopedia of Database Systems, Springer, Boston, MA, 2009. Pp. 1279–1282.
[16] *Regnier M.* Analysis of Grid File Algorithms. BIT - Computer Science Section, 1985. Vol. 25, issue 2. Pp. 335–358.
[17] *Robinson I., Webber J., Eifrem E.* Graph Databases. 2nd ed. O'Reilly, USA, 2015. 237 p.
[18] *Sharma S., Tim U.S., Wong J., Gadia S., Sharma S.* A Brief Review on Leading Big Data Models // Data Science Journal. 2014. Vol. 13, no. 138-157. DOI: https://doi.org/10.2481/dsj.14-041
[19] *Stonebraker M., Brown P., Poliakov A., Raman S.* The Architecture of SciDB. In: Bayard Cushing J., French J., Bowers S. (eds) Scientific and Statistical Database Management. SSDBM 2011. Lecture Notes in Computer Science. Vol. 6809. Springer, Berlin, Heidelberg, 2011. DOI: https://doi.org/10.1007/978-3-642-22351-8_1
[20] Overview of SciDB: large scale array storage, processing and analysis / P.G. Brown // Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 2010. Pp. 963-968. DOI: https://doi.org/10.1145/1807167.1807271

Поступила 11.01.2018; принята к публикации 10.02.2018; опубликована онлайн 30.03.2018.

Об авторах:

Манукян Манук Гарушевич, кандидат физико-математических наук, доцент, Ереванский государственный университет (0025, Республика Армения, г. Ереван, ул. А. Манукяна, д. 1); ORCID: http://orcid.org/0000-0002-8578-3440, mgm@ysu.am

Геворгян Григор Рубенович, кандидат технических наук, Российско-Армянский (Славянский) университет, Республика Армения (0051, Республика Армения, г. Ереван, ул. О. Эмина, д. 123); ORCID: http://orcid.org/0000-0003-1706-568X, grigor.gevorgyan@gmail.com

This is an open access article distributed under the Creative Commons Attribution License which unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).