DOI: https://doi.org/10.15276/hait.03.2021.1
UDC 004.65:519.172

Model and method for representing complex dynamic information objects based on LMS-trees in NoSQL databases

Oleksandr S. Maksymov[1)]
ORCID: https://orcid.org/0000-0001-8951-5251; oleksandr.s.maksymov@onu.edu.ua
Eugene V. Malakhov[1)]
ORCID: https://orcid.org/ 0000-0002-9314-6062; eugene.malakhov@onu.edu.ua. Scopus: ID: 56905389000
Vitaliy I. Mezhuyev[2)]
ORCID: https://orcid.org/0000-0002-9335-6131; vitaliy.mezhuyev@fh-joanneum.at. Scopus: ID: 24468383200
[1)] Odessa National University named after I. I. Mechnikova, 2, Dvoryanska. Odessa, 65082, Ukraine
[2)] FH Joanneum: Kapfenberg, Werk-VI-Straße 46, 8605, Austria

ABSTRACT

The article analyzes the existing approaches to the description of large dynamic information objects in the construction of Automated control systems. Introduced and defined the concept of a Complex Dynamical Information Object. A comparative analysis of the temporal complexities of tree-like structures is carried out and the optimal one for working with Complex Dynamical Information Object is selected. Most modern automated control systems use various approaches to describe automation objects for their operation. Under the automation object, we mean functional objects that are described in the form of structural models that reflect the properties of physical objects. Thus, for optimal work with complex dynamic information objects, we have developed our own model and method for describing the LMS-tree (Log-structured merge-tree), with the ability to split and store down to elementary levels. One of the features of our approach to describing objects is the presence of tree-like levels - the so-called "leaves", by which we mean special tree elements that expand the description of the tree structure of a particular tree level. The minimal elements of the leaves of the tree – "veins" - are details, that is, elementary information elements. A leaf is a combination of "veins" (details) according to certain characteristics, which provide extended information about the level of the tree object. An atomic-level descriptor is a multiple NoSQL database field (array) where the tree level number is the index of the database array. This approach allows you to retrieve and group objects according to the element level of the tree definition.

Keywords: Complex; dynamic; information objects; LMS-trees; NoSQL; models

***For citation*: Maksymov O. S., Malakhov E. V., Mizhuev V. I. Model and method for representing complex dynamic information objects based on LMS-trees in NoSQL databases. *Herald of Advanced Information Technology.* 2021; Vol. 4 No. 3: 211–224. DOI: https://doi.org/10.15276/hait.03.2021.1**

INTRODUCTION

Most modern automated control systems (ACS) for their functioning use a variety of approaches to describing automation objects. We understand the automation object as functional objects, which are described in the form of structural models that reflect the properties of physical objects.

Information about the actual values of a number of parameters of a physical control object is very inaccurate, and the laws of their possible changes are often known only qualitatively.

In this situation, in the process of automating the control of physical objects, we operate with a certain set of properties and characteristics known at the time of designing the automation system. At the same time, the known characteristics do not fully reflect the automation object.

Thus, the concept of Information object appears, which is a view of a physical object from a certain (narrow) position of the control process, i.e.

Information object is a model of some entity of the physical, intellectual or virtual world, which reflects its structure, properties and behavior in the form of information necessary for use in the information system during its functioning.

During the operation of the ACS, changes occur in obtaining more complete information about the object (a set of characteristics is expanded, connections between objects are changed, etc.), while the Information object is complicated. Thus, we can say that the information objects used in the ACS are Complex Dynamical Information Object (CDIO).

For storing CDIO, databases built on the basis of various data models can be used. Despite all the attractiveness, tradition of use and prevalence of classical relational database management systems, they are very limited. This is primarily due to the primitiveness of the data structures underlying the relational data model. Flat normalized relationships are universal and theoretically sufficient to represent

© Maksymov O., Malakhov E., Mezhuyev V., 2021
This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/deed.uk)

data in any domain. However, in unconventional applications, hundreds, if not thousands of tables appear in the database, and expensive join operations are constantly performed on them to recreate the complex data structures inherent in the domain.

Another major limitation of relational systems is their relatively weak ability to represent application semantics. The most that relational DBMSs provide is the ability to formulate and maintain data integrity constraints. Recognizing these limitations and shortcomings of relational systems, database researchers are undertaking numerous projects based on ideas that go beyond the relational data model.

The key factor that made the global IT community think about new strategies for storing and accessing information was the systematic growth of data volumes on the Internet. In this regard, the concept appeared ***Big Data***, which includes some kind of strategy that allows you to efficiently work with huge, constantly growing data sets. And against the backdrop of this concept, the need for a database model that would be more focused on access speed and scalability was clearly looming. Something simpler was needed than the existing relational databases, while not inferior to them in a number of specific tasks. First of all, these are the tasks of building cloud storages, where the end user is primarily concerned with the speed of access and the possible amount of stored information.

NoSQL databases have evolved as the evolution of the relational model, due to the emergence of new requirements for storing and accessing information. NoSQL solutions cannot boast of fundamentally new approaches either – for example, the concept of MongoDB launched in 2008 is a more efficient implementation of the Pick database operation model from 1965. One of the most interesting approaches, in our opinion, is the adaptive database approach, i.e. it is the ability to work with relational and NoSQL database models. A striking representative of this direction is the ADABAS DBMS from Software AG [10].

LITERATURE REVIEW

There are different ways of presenting data to describe information objects when creating an ACS. An incredible number of primary sources are devoted to the discussion of this problem, in which this problem is solved by describing an object in the form of a container. In a sorted associative container, all keys are sorted in a specific order. The simplest example of such a container is a sorted string table (SSTable) [4]. This container is one of the most popular for storing, processing and sharing large datasets. It is used in well-known NoSQL databases such as Cassandra [5], HBase and LevelDB.

A tree-like data structure [9] is a dynamically linked structure in which the relationships between elements are not linear, as in a list, but are like branches of a tree. The simplest tree for describing objects is a binary tree [11]. There are several B-tree implementations [14]. An LSM tree (Log-structured merge-tree) [26] is a data structure that provides a high insertion speed with an acceptable search speed. Another tree-like data structure is the heap [11].

Among the many tree structures used in self-adapting associative containers, you can find the optimal structure for almost any case. When choosing, we were guided by the conditions of the problem (a detailed analysis is presented in [36]). The main condition for choosing a structure is the amount of data that characterizes a particular object.

In our research, we consider the construction of associative data containers to describe complex objects, since they are the most popular and used in NoSQL databases [4]. The methods for constructing key-value data containers can be divided into two categories. One group of methods involves the use of some kind of global ordering (numeric or lexicographic). Keys are stored in a sorted state and a binary algorithm is used for searching. The containers obtained in this way have been called “mixed associative containers”. Examples of such containers are different trees. The second group of methods is hashing, and the containers obtained by this method are called “Hexified Associative Containers”. Examples of such containers are different variants of hash tables.

1. Tree structures

The data of the tree structure [11] is a dynamically linked structure in which the links between the elements are not linear, as in a list, but are similar to the branches of a tree. There are two categories of these structures, which differ in the methods of construction and processing.

The first is “trees” the second is “heaps”. In addition, trees are distinguished by the following characteristics:

Balance. The tree can be:

- degenerative;
- perfectly balanced;
- balanced;
- unbalanced and unexpired.

The number tree of branches. The tree can be:

- binary;
- multipath, when the number tree of branches is more than two.

1.1. Binary trees. The simplest search tree is the binary tree [11]. These trees are among the most popular due to their ease of implementation and very high performance. The main advantage of a binary tree is the ability to implement high-performance sorting and search algorithms built on its basis. In addition, this tree is used in the implementation of the SET and MAP containers in C ++, Treeset and Treemap in Java [12]. Binary trees can be degenerate, balanced, perfectly balanced, or none of these categories. In practice, balanced trees are usually used, since degenerate trees are converted to a list, but often a perfectly balanced laborious construction and balancing in them is sufficient.

There are several B-tree implementations [14].

1.2. AVL tree [15]. In this tree, an additional 2 bits are required for each node of the tree. In addition, when changing a tree, balancing operations are required (when inserting up to two turns, when removing the height of the tree), which also requires additional costs.

1.3. Red-black tree [16]. When implementing this tree, it is necessary that each vertex preserves color (1 bit). Sometimes, due to the need to align memory, this condition leads to large memory consumption. In addition, when changing a tree, balancing operations are required (when inserting up to two turns, when deleting up to three), which also requires additional costs. Red-black trees are used in various fields: in the Linux kernel for queuing, in the ext3 filesystem, and in various libraries for implementing SET and MAP.

In addition, there is a modification of the AA tree, in which there is one more condition: the red node can only be the right child. This condition makes it possible to simplify the execution of all operations, since there are fewer possible cases for parsing. The speed of this tree can be compared to a red-black tree, but the AA-tree for each node also preserves a “level”, which leads to additional memory costs.

Red-black trees can be 1,388 times the height of AB for the same number of nodes. This leads to the fact that the insertion and search times can be longer in the red-black tree, but removing from the AVL tree may require a number of iterations equal to the tree depth, making it more advantageous to remove it in red-ebony.

1.4. Cartesian tree [17]. This structure also has other names such as Deramida ((Structure Treap),) as this data structure is a binary tree and pyramid association. Pyramid – a data structure similar to a tree, but with one condition: the value at any node is not less than the value of its children. Each node of the access tree contains a pair of values (x, y), where X is called a key, y is a priority. Thus, on this tree, a heap is obtained, and on x – a binary tree.

Priorities are usually chosen randomly to avoid excessive tree heights. Two operations are used to perform basic tree transactions: merge and split.

It should be noted that the worst-case average of some operations on one tree may take longer, but then other operations will be completed in less time. It should also be noted that it is difficult to create such a tree.

Another peculiarity of the tree is a rather large excess of memory (2 to 4 bytes per node for storing height). Thus, this tree cannot be used where performance is guaranteed, such as in real-time systems and OS kernels.

These trees are well suited for collecting statistics on a large number of parameters, since these trees allow us to count the number / difference, minimum / maximum, and other operations in a reasonable amount of time.

In addition, there is a modification - a Cartesian tree by an implicit key. This structure provides a dynamic array interface, but it is implemented using a Cartesian tree. The Cartesian tree for the implicit key allows you to implement a large number of operations with an array and a subarray in logarithmic time.

1.5. Splay tree [18] – a binary tree that has no additional requirements for the tree structure and is balanced in the process. Some points in the Splay tree can be completely unbalanced. The Splay tree maintains balance with a dedicated Splay feature. This function searches for the required node and makes it root using simple rotations that change the structure of the tree, while maintaining the order in the tree. Splay is activated after every operation, including search.

In this tree, the complexity of operations can be guaranteed, since the tree may be unbalanced when requested, therefore, the height of the tree is greater than log (n). This score is achieved in that finding items that have been used recently will be faster and will compensate for the earlier case.

The main advantage of this tree is productivity. Testing in real conditions and the expanded tree turned out to be one and a half to two times more efficient than other balanced binary trees. However, it is theoretically proven that if the probabilities of using the elements are the same in time, then the tree will work in the same way as other implementations. At the same time, for arbitrary probabilities, access

to tree elements works one and a half times slower. Thus, the extensible tree can be used in real-time systems and in most general-purpose libraries.

In addition to performance, the advantages of this tree include the absence of additional memory costs, and the disadvantages are a strong limitation when used in a multimedia environment and limitations in purely functional languages.

1.6. Scapegoat tree [19]. This tree is an alpha parameter that is in the range (0.5, 1). This parameter is specified when creating a tree and determines the height of the tree.

Therefore, Alpha affects the speed of modification and search, for example, with Alpha = 0.5 we get a balanced tree (maximum search speed, but minimum modification speed), and with Alpha → 1 we get a link (maximum modification speed, but minimum search speed). In this case, the search is similar to a regular search tree, but insertion and deletion are different.

To implement these operations, an element that unbalances is searched for, and then the base slows down, where the root is the element that unbalanced. In the worst case, the modification operations can take N times (depending on the node), but this operation is distributed over the tree, so the average time for the worst case (Log N).

2. Multipath trees

In addition to binary trees, there are trees with more than two branches. Such trees are called multipath or highly branching trees. The most widespread is the B-tree and its various modifications [18]. The B-tree is used in database systems (indexes in many modern database management systems) and file systems. There are many variations in this tree. The most famous of them are presented below.

2.1. B + -Tree also known as the Bayer-Baum tree [19]. This tree is used in file systems for storing directories and indexing metadata (NTFS, BEFS, etc.). In relational DBMS – as an index (Oracle, SQLite, etc.), NoSQL databases – for data access (CouchDB).).

2.2. B * -tree [22**].** This data structure is similar to a b-tree, but has a different minimum vertex fill factor – ⅔ (in a b-tree). This modification allows for more efficient use of memory and provides a small performance gain. The disadvantages of B * – trees include a more complex function of separating crowded nodes. The uses of this tree are similar to those of the b-tree.

2.3. 2-3- Wood [11]. This tree is a separate case of a B + tree, which can have nodes with only one key (contains the maximum left subtree) and two descendants and nodes with two keys (containing the maximum left and middle subtrees) and 3 children. Terminal nodes have no children. In addition, there are 2-3-4-trees (B-Trees of degree 4), built according to similar rules. It can be used to store dictionaries in internal memory to avoid cache penalties.

3. Multipath distribution trees are optimized for write

For a long time, the B-tree has no alternatives among data structures for storage in external memory. But recently, the situation has begun to change due to the need to process the ever-growing volumes of data. To achieve this, data structures are optimized specifically for write operations. Some of these structures are partially inferior to the tree in terms of search speed, but allow more efficient production and deletion.

There are several types of data structures:

- trees using a buffer;
- LSM and fractal trees

Buffer Trees

Buffer trees can also be divided into several types:

3.1. Buffered tree [23] – (a, b) - tree with coefficients a = m / 4, b = m, built over a set of N leaves, each of which contains b elements. In this case, each internal node contains a buffer of m elements. The data is stored only in the letter, only individual keys are located in other nodes. The data is first added to the buffer of the root node and only after the overflow of this node is allocated to buffers of subsequent levels, etc. This happens until the data reaches the lowest level, at which it already remains. In this case, a reduction similar to a tree is used if necessary. Using a buffer allows you to reduce the insertion time due to batch processing of requests, but the use of autonomous models (the answer to any request does not come immediately, but after a while) makes it unavailable to use the buffered tree for tasks, where the answer is required immediately and further depends on it logics.

Based on this tree, a priority queue has been implemented: Range Tree and Segment Tree, which successfully solve some geometric problems and problems on graphs. In addition, this tree allows for high-performance memory sorting. It is not necessary to have all the elements before sorting.

3.2. Bε-trees [24] are a class of trees that differ from buffered trees by finding the data and the presence of the ε parameter. The search operation in a Bε-tree is also similar to a search in a buffered tree, but requires an immediate response, which, in turn, requires additional operations to search for data in the buffers.

The parameter ε determines the size of the buffer (≈ b - Bε) and the size of the keys in the node (≈ bε) during tree initialization and varies from 0 to 1.

For ε = 1, the usual B-tree (buffer size 0), and for ε = 0, a modification of the buffered tree – Buffered Repository Tree [25]. It is also an (a, b) – tree, but with coefficients a = 2, b = 4 and buffer size B instead of m. This tree is widely used for width and depth traversal of a graph.

4. Heap

Another tree-like data structure is the heap [9]. This container is widely used for various tasks. Based on the purchase of a queue with priority and heapsort, various search algorithms and graph algorithms are used [31].

There are several purchased heaps. The main ones are listed below.

4.1. Binary heap [32], also known as a pyramid. It is a binary tree for which the main heap property is met. All levels in this heap must be filled, except, perhaps, the last level, which must be filled from left to right.

4.2. Binomial heap [32].This data structure is a sorted set of binomial trees, each of which is a property of the heap. In addition, all trees are different in size. A binomial tree is defined inductively. B_0 contains only one node, B_k contains two binomial trees B_{k-1}. In this case, the root of the first tree is a descendant of the root of the second tree.

4.3. Fibonacci heap [33]. This structure consists of many Fibonacci trees (these are n-trees for which the main property of the heap and the top of one layer are associated with a doubly linked list). The call to buy is a link to the minimum element contained in the root of one of the trees.

This heap has good performance for all operations except delete.

4.4. Thin pile [34] presents a set of thin trees that satisfy the properties of the heap (while the ranks of the trees can be repeated). A thin tree TK of rank K is called a binomial tree B_k in which the remaining descendants at several vertices are removed. Moreover, it is impossible to delete at the end nodes (since they have no descendants) and near the root (otherwise, a binomial tree with a lower rank). In practice, a thin beam is used to implement the priority queue, but it is more efficient than Fibonacci, since it has smaller constants in the operating time.

4.5. Thick pile [34] is a set of thick trees in which the ranks can be repeated, and there can be no more than two nodes per rank containing a value less than the value of the ancestor.

Thick wood is determined inductively. F_0 contains one vertex, F_k contains three ranks k-1, and near the root of one of them, the leftmost sons are the roots of the other two.

A thick heap, like a thin heap, is a Fibonacci modification, but in practice requires less space and is more efficient.

4.6. Left heap [35] – A data structure based on a binary left tree for which the main heap property is satisfied. In this case, the tree can be unbalanced. In addition, for this purchase, the condition must be met: the closest place to insert the vertex must be the rightmost position in the tree. If we denote D (v) as the distance from vertex v to the nearest place for insertion, then for all vertices it is necessary to execute D (v.left)> = D (v.right). Since there is no task in this pile, it is persistent and can be implemented in a functional language.

4.7. 2-3 Heap [36] are a set of trees h(i), where i = 0, .. , n, for which the main heap condition is met. h(i) is a tree from zero to two 2-3-trees of degrees I, which are combined into the following rule: the root of the skin becomes the rightmost son in the previous one. The 2-3 tree is marked as induced. T_0-tree from one vertex, T_j-two or three T_{j+1} trees, which are combined according to the same rule as h(i), The specified device of 2-3 heaps allows balancing both with the selection of elements, and with the addition, which makes it possible to increase the efficiency of extracting the minimum in comparison with Fibonacci.

4.8. Priority queue for Brodal and Okasaki [35].

This data structure is built on several principles:

- Application of a special asymmetric embankment. This allows you to insert an element in a short time.
- Storage is minimal, which allows you to quickly retrieve it.
- Data-Structural self-tuning idea that allows you to keep the queue in the queue and reduce the number of merges.

Thus, the Brodal and Okasaki priority queue consists of a minimum and special priority queue, which stores the Brodal and Okasaki priority queue, sorted by T-min. This description can be represented in the formula BPQ = <Tmin, PQ (BPQ)>, where BPQ stands for Bootstrapping Priority Queues. As a priority queue (PQ), a special stack is used - asymmetric. This idea is based on the use of

asymmetric binary coding of a number using the digits 2 in the last disarmed bit, which allows you to have a maximum of one tree of one binomial rank, with the exception of the minimum rank (2 trees).

This enhancement to the binomial heap avoids cascading tree merges during insertion and reduces insertion time.

LSM and fractal TREE

LSM tree (Log-Structured Merge-Tree) [26] – a data structure that provides fast insertion speed with acceptable search speed. This tree is also known as a log-structured merge tree because it is well suited for storing logs of various operations that are constantly updated and reviewed frequently.

This tree consists of two or more structures, each of which is highly efficient on the device on which it is stored. In the simplest case, an LSM tree contains two tree structures that differ in size.

The smaller one is in the internal memory, the larger one is in the external one.

In this case, the insert is only done into the smaller tree (since it is in RAM and much faster), and when it reaches a certain size, the tree from internal memory is sent to external memory and merged with the larger tree by merging.

In practice, LSM trees are usually used, which have several levels. In this case, each level is represented by a tree structure, and in case of overflow, a combination with the next level.

The LSM tree is widely used in NoSQL databases such as Apache Cassandra, BigTable, Leveldb and many others, as well as in the new Disk Engine tool for Tarantool. These trees are especially effectively used for data with varying degrees of relevance (message feed, chats, walls in social networks, events), storage of timeseries and logs.

The main disadvantage of the LSM tree is the need to search at each level with high cost. This is how another tree appeared – a fractal one [30]. It was originally based on the COLA (Cache-Oblivious Lookahead Array) architecture, but now it is a modification of the Bε-tree with various performance improvements and a 4MB tile size, which is significantly larger than the normal tile size -wood. It uses a fractal cascade to lower search costs.

The basic idea is that when looking for Ti, we know where the key should be at this level, and we can use this information to improve the search at the next level Ti + 1. To do this, links to the following levels are added to the links. LSM trees and an in-tree analogue is obtained. In addition, each node is assigned a buffer in which all changes are made.

In a fractal tree; there are tree nodes, node elements and buffers. When the buffer overflows, it is sent to the child buffers until it reaches the leaves where the elements are being filled. In this case, all non-leaf nodes serve as indexes for the search, and the leaf already contains information.

In such a scheme, insertion is exactly fast (since the parent with multiple buffers is always in RAM) and the lookup takes the same amount of time as in-tree.

The main disadvantage in comparison with the LSM tree is the complex sequential reading, as well as the rather difficult deletion and updating of information, as it is necessary for data retrieval.

THE PURPOSE OF THE ARTICLE

The purpose of this work is to develop methods for increasing the flexibility of description and processing of complex dynamic information objects in automated management systems by presenting them on the basis of LMS trees in NoSQL databases.

To achieve this goal, the following tasks were solved in the work:

Possibilities of increasing the speed of processing the CDIO by means of their representation using dynamic structures in NoSQL are analyzed.

A model and a method for representing LMS using LMS trees using dynamic structures in NoSQL have been developed.

The increase in the processing speed of CDIO is shown on the example of representation in the NoSQL ADABAS DBMS.

MAIN PART. ANALYSIS OF APPROACHES TO THE DESCRIPTION OF CDIO

If we consider CDIO as an object for its basic processing (storage, updating, deletion), as well as the use of distributed processing, then we need to compare the time spent on these actions when using tree structures.

Comparative characteristics of the time spent are shown in the summary table (Table 1) of the time complexity of tree structures.

Among the many tree structures used to describe complex dynamic objects of information systems, the optimal structure can be found for almost any case. If you choose a problem condition (see [38] for a detailed analysis). The main condition for choosing a design is its clarity and ease of interpretation.

Table 1. **Summary table of time complexities of tree structures**

Submitted	Quality / Operator	Positioning	Search	Insert	Deleting
B-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (n)	O (n)	O (n)
AVL-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (log n)	O (log n)	O (log n)
RB-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (log n)	O (log n)	O (log n)
D-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	amortized O (log n)	amortized O (log n)	amortized O (log n)
Splay-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	amortized O (log n)	amortized O (log n)	amortized O (log n)
Scapegoat-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (n)	amortized O (log n)	amortized O (log n)
MP-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (log n)	O (log n)	O (log n)
AVL-tree	The average	O (n)	O (log n)	O (log n)	O (log n)
	Worst	O (n)	O (log n)	O (log n)	O (log n)
Buff-tree	The average	O (n)	O $((\log_m n) / B)$	O $((\log_m n) / B)$	O $((\log_m n) / B)$
	Worst	O (n)	O $((\log_m n) / B)$	O $((\log_m n) / B)$	O $((\log_m n) / B)$
Bε – tree	The average	O (N / B)	O $(\log_b N)$	O $(\log_b N/\sqrt{B})$	O $(\log_b N/\sqrt{B})$
	Worst	O (N / B)	O $((\log_b N))$	O $(\log_b N/\sqrt{B})$	O $(\log_b N/\sqrt{B})$

Source: **compiled by the authors**

In case the interpretation continues in memory, you can use balanced binary trees, the choice of which should be based on many factors: complexity of construction, additional memory, product performance factor, and deletion. If the problem uses multidimensional descriptions, it is possible to use special trees to provide multidimensional information [39]. If there is a need to use a container as a priority queue or to solve a minimum / maximum search task, it is possible to use a bundle (any option depending on the needs of efficiency and consistency).

In case the data does not fit in RAM and you need to use external memory, you must use other trees. If you are using multidimensional data, you need to use special trees to provide multidimensional information using external memory [39].

In other cases, the underlying data structure is a B-tree (or a variation of it, depending on the need for sequential key access, additional memory, and implementation complexity). If necessary, the best choice would be a large number of records – using record-optimized trees (LSM trees, fractal and buffered).

Thus, after performing the analysis, we see that the LSM tree provides a sufficient insertion speed with an acceptable search speed.

DEVELOPMENT OF A MODEL AND METHOD REPRESENTING CDIO USING LMS-TREES

Thin complex dynamical systems, considered as control objects, have the following basic properties (system factors) [3]:

- integrity and the possibility of decomposition into elements A (objects, subsystems);
- the presence of stable links (relations) R between the elements of A;
- arrangement of elements in a certain tree structure (Str);
- providing elements with parameters (P);
- the presence of synergistic (integrative) properties Q, which no one possesses in the elements of the system;

- multiple laws, rules and operations Z with the above attribute systems;
- having a goal of functioning and development (G).

Thus, the system is a set (1)

$$Syst = \{A, Str, Q, R, Z, G\}. \quad (1)$$

In this work, we consider one of the components of complex systems – this is element A (objects and subsystems), which must be described, technologies and tools for their creation, storage and use, and also show their interconnections and interactions.

To describe the object, we will use the LSM-tree (Log-structured merge-tree),

One of the features of our approach to describing objects is the presence of tree-like levels - the so-called "leaves", by which we mean special tree elements that expand the description of the tree structure of a particular tree level. The minimal elements of the leaves of the tree – "veins" - are details, that is, elementary information elements. A leaf is a combination of "veins" (details) according to certain characteristics, which provide extended information about the level of the tree object.

Thus, in order to build a model of CDIO, we can apply the statements of the Algebra of Sets – since our object is described by a certain set of properties or a set of properties, these properties characterize each level uniquely, and are supplemented by the properties of subordinate levels, i.e. we have the intersection (product) of sets of properties.

In this case, the intersection is obtained from the top-level properties. When it comes to filling levels, the intersection is inherited from the top level, and the symmetric difference is filled at this level.

The CDIO model can be represented as follows (2):

$$A=\bigcup_{i=1}^{n} x_i , \quad (2)$$

where xi={$S_1S_2...S_n$};
x_i – this is a set of levels of CDIO; S_i -is a lot.

$$S_i=\bigcup_{j=1}^{m} p_j , \quad (3)$$

where p_j properties of CDIO levels.

These statements make it possible to determine the method of forming the LSM-tree of the description of the CDIO.

An important issue after the description of the object is the storage technology and, accordingly, access (add, delete, update). To the stored data for the effective operation of the information system based on the tree-like description of the object.

Having overlapping sets of level properties, based on multiple data description elements in the NoSQL DBMS, we get a tool for storing, accessing and updating CDIO descriptors. At the same time, we have the ability to accelerate access to the elements of the LSM-tree, due to the use of associative search methods in multiple data structures using NoSQL of the ADABAS DBMS.

REPRESENTATION TO CDIO WITH NoSQL ADABAS

To present the description of the LED, we will use a specific Automated System "DivMiks", which functions to automate the work of company selling furniture. An example of a CDIO will be the document "Receipt invoice" for performing the operation "Posting to the furniture warehouse".

The initial description is a three-level tree, on the first level there is the "Company Supplier of the Goods", on the second level there is the "Invoice Header" (this is the part of the document that describes its main details) and the third level is the "Item Nomenclature Table". Thus, we need to complete the description of all these elements and store it all in the database.

For this we use the NoSQL ADABAS DBMS. This DBMS has tools for multidimensional data representation – these are multiple fields (data arrays) and periodic groups (arrays of data structures) (Fig. 1).

The periodic group stores "Leaves", their representations in the database allow storing extended information on the description of the object.

To write program code, we use the Natural development environment, which is the 4GL Natural programming language.

To communicate with the database, this development environment uses a special data description module (DDM), which for our case looks like this (Fig. 2).

Fig. 3, using the DivMix information system as an example, shows the description of the "Shipping waybills" object in the form of tree and leaf elements.

Sheet details are dynamically generated based on the named configuration. The configuration is stored in the NoSQL database in the form of a multidimensional table, which has the following structure (Fig. 4).

A database file is a multidimensional cube into which an XML file with the specified configuration is loaded.

An example of a named configuration file is shown in Fig. 5.

C6	1		Periodic Group	
C7	2	32	Alpha	Descriptor, Null Value Suppression
CA	2	50	Alpha	Descriptor, Null Value Suppression
CE	2	10	Alpha	Descriptor, Null Value Suppression
C9	2	168	Alpha	Descriptor, Null Value Suppression
CB	2	10	Alpha	Descriptor, Null Value Suppression
CF	1	200	Alpha	Descriptor, Multiple-Value, Null Value Suppression
D6	1	32	Alpha	Descriptor

Fig. 1. **Description of the periodic group (C6) and multiple field (CF) in the ADABAS DBMS**
Source: compiled by the authors

Type	Level	Short Name	Name	Format	Length	Suppression	Descriptor
	1	A1	DOC_КОДПОСТАВЩИКА	A	32		D
	1	A2	DOC_НАИМЕНОВАНИЕ	A	60		D
G	1	A3	DOC_ДЕРЕВО				
	2	A4	DOC_РОДИТЕЛЬ	A	32	N	D
	2	A5	DOC_УРОВЕНЬ	N	8.0	N	D
	2	A6	DOC_ТИПУРОВНЯ	A	2	N	D
	2	A7	DOC_ИМЯУРОВНЯ	A	120	N	D
	2	A8	DOC_КОДУРОВНЯ	A	32	N	D
	2	A9	DOC_ПУТЬ	A	200	N	D
	1	BE	DOC_ДАТА-СТАРТ	N	8.0		D
	1	BF	DOC_ДАТА-КОНЕЦ	N	8.0		D
	1	C0	DOC_ТИПДОКУМЕНТА	A	2		D
	1	C1	DOC_ПУТЬДОГОВОР	A	200		
	1	D4	DOC_ДОГОВОР	A	60		D
	1	D5	DOC_ТЕКСТ	A	160		
	1	C8	DOC_STATUS	N	2.0		D
	1	C5	DOC_ISN-RET	N	6.0		D
P	1	C6	DOC_CONTENT				
	2	C7	DOC_POLE	A	32	N	D
	2	CA	DOC_NAME	A	50	N	D
	2	CE	DOC_EIZ	A	10	N	D
	2	C9	DOC_ZNACH	A	168	N	D
	2	CB	DOC_ZAPOLNEN	A	10	N	D
M	1	CF	DOC_PATH	A	200	N	D

Fig. 2. **Description of Natural data structures**
Source: compiled by the authors

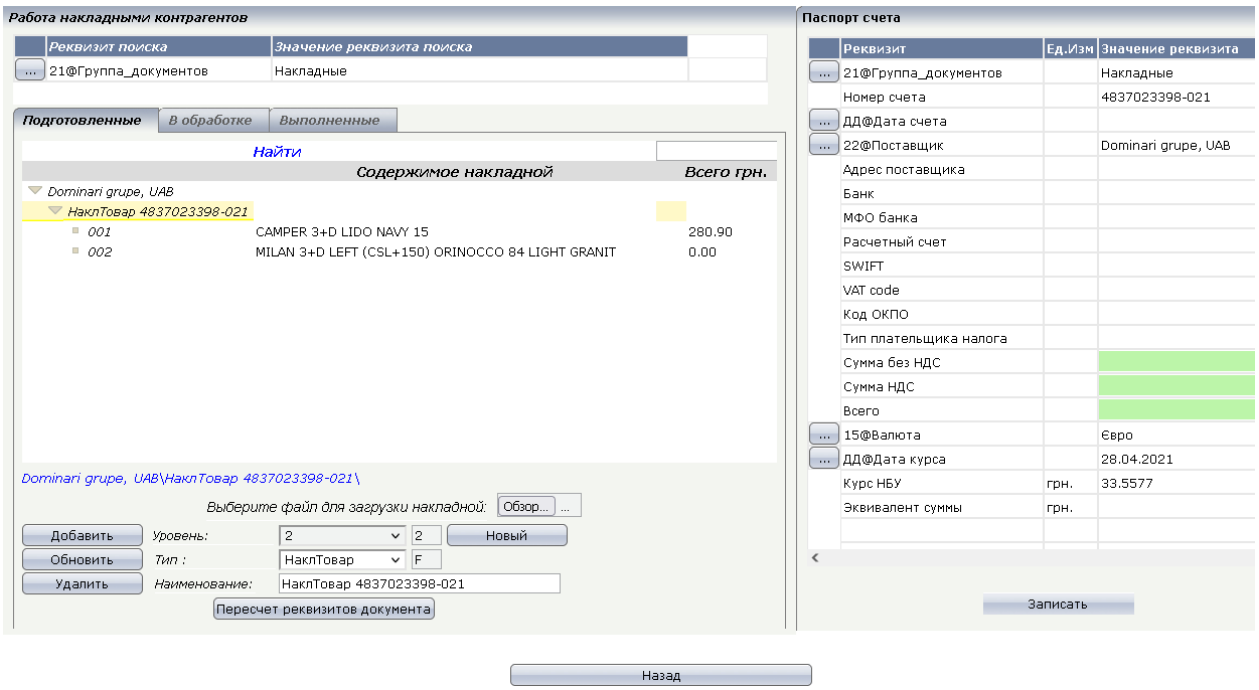


Fig. 3. **An example of the appearance of a tree-like description of an object with leaves**
Source: compiled by the authors

Type	Level	Short Name	Name	Format	Length	Suppression	Descriptor
	1	A1	KLFILE	A	12		D
	1	A6	KLFILIAL	A	3		D
P	1	A2	KLSPRAV				
	2	A3	KLPOLE	A	32	N	D
	2	A4	KLZNAC	A	240	N	D
	1	A5	KLVIBOR	A	252		S
	1	B0	KLVIBORI	A	24		S
	1	B1	KLFROM3	A	74		S
	1	B5	KLDEBET	A	26		S

Fig 4. **The structure of the NoSQL database file for storing the configuration**
Source: compiled by the authors

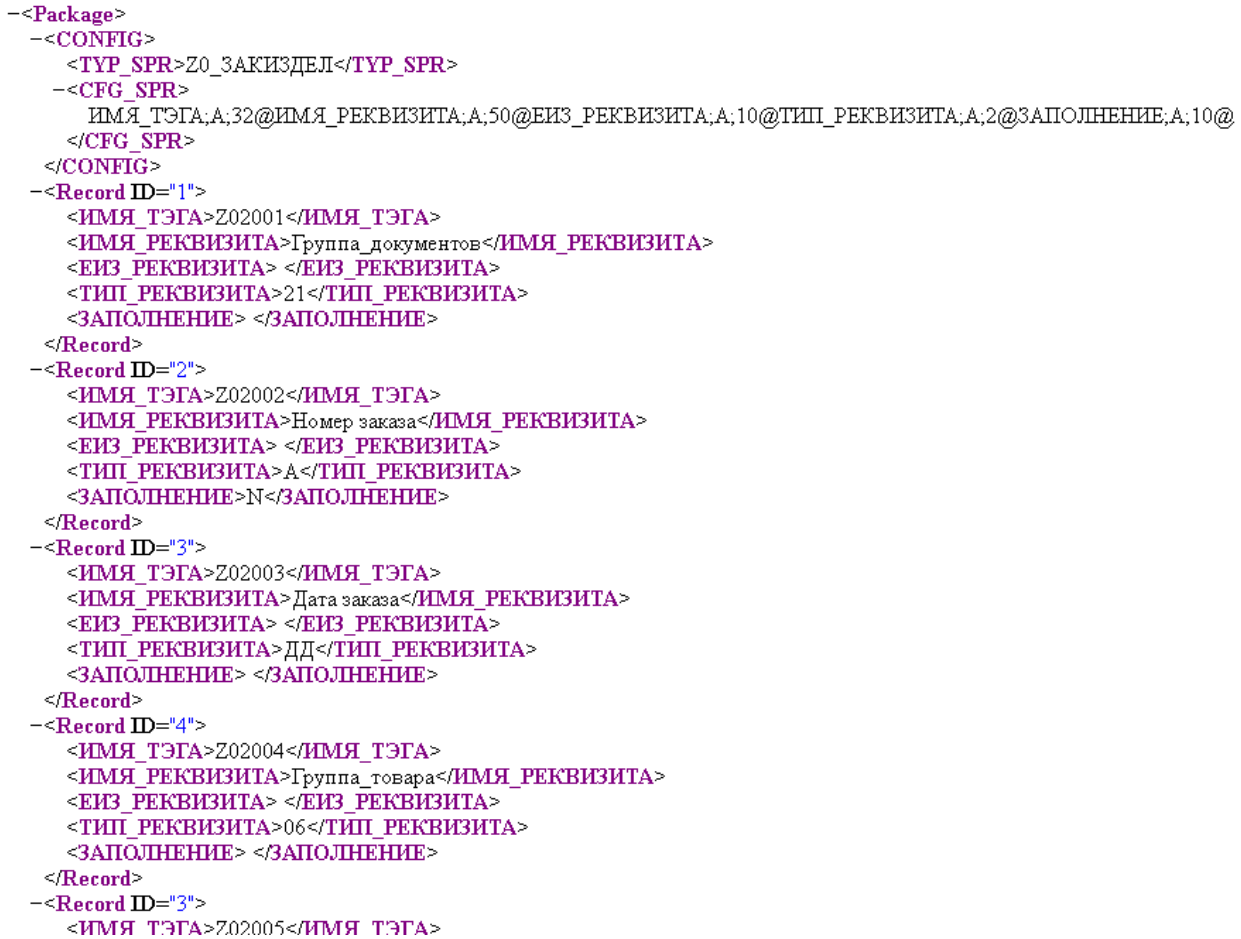
CONCLUSIONS

Thus, for optimal work with complex dynamic information objects, we have developed our own method for describing a tree with the ability to split and store down to elementary levels. An atomic-level descriptor is a multiple NoSQL database field (array) where the tree level number is the index of the database array. This approach allows objects to be retrieved and grouped according to the element level of the tree definition, which provides quick access to data as well as tree-level extensions called “leaves”.

This approach to the description of objects allows you to get an effective technology for working with unstructured data, describe them and provide the ability to build adaptive information systems.

The flexible and dynamic environment of modern information systems constantly requires changes in the description of control objects. There are restrictions on data descriptions for specific models. Our approach – a tree-like representation of data with a flexible technology for the formation and storage of objects, allows us to form a new platform for building modern information systems with an effective tool for accessing large amounts of data.

Our technology solves another problem – storage redundancy when building information systems. When changing the structure of information in the data warehouse, it is necessary to change the



```
−<Package>
  −<CONFIG>
     <TYP_SPR>Z0_ЗАКИЗДЕЛ</TYP_SPR>
    −<CFG_SPR>
       ИМЯ_ТЭГА;A;32@ИМЯ_РЕКВИЗИТА;A;50@ЕИЗ_РЕКВИЗИТА;A;10@ТИП_РЕКВИЗИТА;A;2@ЗАПОЛНЕНИЕ;A;10@
     </CFG_SPR>
   </CONFIG>
  −<Record ID="1">
     <ИМЯ_ТЭГА>Z02001</ИМЯ_ТЭГА>
     <ИМЯ_РЕКВИЗИТА>Группа_документов</ИМЯ_РЕКВИЗИТА>
     <ЕИЗ_РЕКВИЗИТА> </ЕИЗ_РЕКВИЗИТА>
     <ТИП_РЕКВИЗИТА>21</ТИП_РЕКВИЗИТА>
     <ЗАПОЛНЕНИЕ> </ЗАПОЛНЕНИЕ>
   </Record>
  −<Record ID="2">
     <ИМЯ_ТЭГА>Z02002</ИМЯ_ТЭГА>
     <ИМЯ_РЕКВИЗИТА>Номер заказа</ИМЯ_РЕКВИЗИТА>
     <ЕИЗ_РЕКВИЗИТА> </ЕИЗ_РЕКВИЗИТА>
     <ТИП_РЕКВИЗИТА>A</ТИП_РЕКВИЗИТА>
     <ЗАПОЛНЕНИЕ>N</ЗАПОЛНЕНИЕ>
   </Record>
  −<Record ID="3">
     <ИМЯ_ТЭГА>Z02003</ИМЯ_ТЭГА>
     <ИМЯ_РЕКВИЗИТА>Дата заказа</ИМЯ_РЕКВИЗИТА>
     <ЕИЗ_РЕКВИЗИТА> </ЕИЗ_РЕКВИЗИТА>
     <ТИП_РЕКВИЗИТА>ДД</ТИП_РЕКВИЗИТА>
     <ЗАПОЛНЕНИЕ> </ЗАПОЛНЕНИЕ>
   </Record>
  −<Record ID="4">
     <ИМЯ_ТЭГА>Z02004</ИМЯ_ТЭГА>
     <ИМЯ_РЕКВИЗИТА>Группа_товара</ИМЯ_РЕКВИЗИТА>
     <ЕИЗ_РЕКВИЗИТА> </ЕИЗ_РЕКВИЗИТА>
     <ТИП_РЕКВИЗИТА>06</ТИП_РЕКВИЗИТА>
     <ЗАПОЛНЕНИЕ> </ЗАПОЛНЕНИЕ>
   </Record>
  −<Record ID="3">
     <ИМЯ_ТЭГА>Z02005</ИМЯ_ТЭГА>
```


Fig 5. **An example of a named configurator**
Source: compiled by the authors

physical structure, and this is one of the problems of modern automation systems. To solve this problem and build a new generation of systems, we use the NoSQL data model, which is an extended relational model that removes the restriction on the indivisibility of data stored in table records. A set of values for multiple fields is considered an independent table embedded in the main table. The main advantage is the ability to represent a collection of related tables with a single NoSQL table. This provides high visibility and improved information processing.

The software platforms developed using this technology provide minimal costs for the support and development of such automated systems, as well as reduce time resources, which allows a minimum number of employees to support a larger number of software systems.

REFERENCES

1. Malakhov, E. V. “Selection of the complex-structured subject domains”. *Materials of the International Scientific and Technical Conference* “*Modern methods, information, software and technic support of the management systems of organizational and technological complexes*”. *Publ. NUFT*. Kyiv: Ukraine. 26-27 November, 2009. p. 79–80.

2. Musser, D., Durge, J. & Sainey, A. “C ++ and STL. Reference Guide.” *Publ. Williams*. Moscow: Russian Federation. 2010.

3. Sadalj, P. J. & Fowler, M. “NoSQL: A New Methodology for the Development of Non-Relational Databases”. *Publ. Williams*. Moscow: Russian Federation. 2013. 172 p.

4. “SSTable & LSM-Tree”. – Available from: http://www.mezhov.com/2013/09/sstable-lsm-tree.html. – [Accessed 13.09.2020].

5. Carpenter, J. & Hewitt, E. “Cassandra: The Complete Guide”. *Publ.O'Reilly Media.* New York: USA. 2016. 360 p.

6. “SSTable and log-structured storage: LevelDB”. – Available from: https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb. – [Accessed: 13.09.2020].

7. Aho, A. V. Hopcroft, D. E. & Ullman, D. D. “Data structures and algorithms”. *Publ. Williams.* Moscow: Russian Federation. 2003. 382 p.

8. Kormen, T. Kh. [et al.] “Algorithms: construction and analysis”. *Publ. Williams Publishing House.* Moscow: Russian Federation. 2005.1296 p.

9. Topp, W. & Ford, W. “Data structures in C ++”. *Publ.Binom.* Moscow: Russian Federation. 2000. 815 p.

10. “Collaborate with Adabas & Natural community experts”. [Electronic resource]. – Available from: https://techcommunity.softwareag.com/en_en/adabas-natural.html. – [Accessed: 13.09.2020].

11.Wirth, H. “Algorithms + data structures = programs.” *Publ. Mir.* Moscow: Russian Federation. 1985. 406 p.

12.“Data structures: binary trees”. Part 2. “An overview of balanced trees”. [Electronic resource]. – Available from: https://habrahabr.ru/post/66926/. – [Accessed: 13.09.2020].

13. Knut, D. E. “The art of computer programming.” *Publ. Mir*. Moscow: Russian Federation. 1976. 736 p.

14. Sedgwick, R. “Fundamental algorithms in Java.” Part 1-4. “Analysis. Data structures. Sorting. Search.” *Pabl. DiaSoft.* Kyiv: Ukraine. 2002. 688 p.

15. Aragon, C. R. & Seidel, R. “Randomized Search Trees, Proc. 30th Symposium Fundamentals of Computer Science”. *Publ. DC: IEEE Computer Society Press.* Washington: USA. 1989. p. 540–545.

16. Sleator, D. D. & Tarjan, R. E. “Self-Tuning Binary Search Trees“. *ACM Journal*. 1985; No. 32: 652–686.

17. Andersson, A. “Improving Partial Reorganization Using Simple Balance Criteria. Tr. Workshop on algorithms and data structures”. *Publ. Springer-Verlag.* Berlin: Germany. 1989. p. 393–402.

18. Levitin, A. V. “Algorithms. Editorial introduction and analysis”. *Publ. Williams.* Moscow: Russian Federation. 2006. 576 p.

19. Zubov, V. S. & Shevchenko, I. V. “Structures and methods of data processing. Workshop in Delphi environment”. *Publ. Filin.* Moscow: Russian Federation. 2004. 304 p.

20. Berliner, H. “A tree search algorithm B*. Best First Evidence Procedure”. *Artificial Intelligence*. 1979; No. 12: 23–40.

21. Ardzh, L. “Buffer Tree: A New Method of Optimal I. Algorithms in Sb. Workshop on algorithms and data structures.” *Publ. Springer-Verlag.* Berlin: Germany. 1995. p. 334–345.

22. Bender, M., Farach-Colton, M., Jannen, V., Johnson, R., Kushmaul, B. C, Porter, D. E., Yuan, J. & Zhang, Yu. “Introduction to Bε-trees and record optimization”. *In: login; Magazine.* 2015; Vol. 40 No. 5: 22–28.

23. Buchsbaum, A. L., Goldwasser,, M., Venkatasubramanian, S. & Westbrook, J. R. ”On external memory graph traversal” . *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00). Society for Industrial and Applied Mathematics*. Philadelphia: USA. 2000. p. 859–860.

24. O'Neil PE, Cheng E, Gawlick D & O'Neil EJ. “Log-structured merge tree (LSM-tree)”. *Acta Informatica.* 1996; Vol. 33 No. 4: 351–385.

25. Sears, R. & Ramakrishnan R. “bLSM: A fusion tree with a general purpose logical structure”. *Proceedings of the ACM SIGMOD 2012 International Conference on Data Management*. NY: ASM. 2012. p. 217–228.

26.Tan, V., Tata, S., Tan, Y. & Fong, L. “Differential Index: Differential Index in Distributed Storages of Log-Structured Data”. *Proceedings of the 17th International Conference on the Expansion of Database Technology, EDBT. OpenProceedings.org*. Konstanz: Germany. 2014. p. 700–711.

27. Bender, M. A., Farach-Colton, M., Fineman, J., Vogel, J., Kuzmaul, B. & Nelson J. “Streaming B-trees without caching”. *Proceedings of the 19th ACM Annual Symposium on Parallelism in Algorithms and Architectures*. *Publ. ACM Press*. CA. 2007. p. 81–92.

28. Alekseev, V. E. & Talanov, V. A. “Graphs and algorithms. Data structures. Computation models”. *Publ. Internet Un-t Inform. Technology BINOM*. Moscow: Russian Federation. 2006. 320 p.

29. Cormen, T. H. [et al.]. “Algorithms: Construction and Analysis = Introduction to Algorithms”. *Publ. Williams.* Moscow: Russian Federation. 2005. 1296 p.

30. Fredman, M. L. & Tarjan, R. E. “Fibonacci heap and their use in improved network optimization algorithms”. *Journal of the Association for Computational Engineering*. 1987; Vol. 34 No. 3: 596–615.

31. Kaplan, H. & Tarjan, R. E. “Thin heaps, thick heaps”. *ACM Algorithm Transactions (TALG).* 2008; Vol. 4 Article No. 3: 1–14.

32. Clark, A. “Linelists and Priority Queues as Balanced Binary Trees”. *Stanford University*. Stanford: 1972.

33. Takaoka, T. “The theory of 2-3 heaps”. *Discrete Applied Mathematics*. 2003; Vol. 126: 115–128.

34. Brodal, G. S. & Okasaki, S. “Optimal purely functional priority queues”. *Journal of functional programming*. 1996; Vol. 6: 839–857.

35. Potapov, D. R., Artemov, M. A. & Baranovsky, E. S. “Review of adaptation conditions for self-adapting associative data containers”. *Bulletin of the Voronezh State University. Series: System Analysis and Information Technology.* 2017; No. 1: 112–119.

36. Gulakov, V. K. & Trubakov, A. O. “Multidimensional data structures”. *Publ. BSTU.* Bryansk: Russian Federation. 2010. 387 p.

37. Knut, D. “The art of programming. Vol. 3. Sorting and search “.*Publ. Williams.* Moscow: Russian Federation. 2007. 824 p.

38. Pug, R. & Flemming, F. R.”Cuckoo hashing”. *Journal of Algorithms*. 2004; Vol. 51: 122–144.

39. Fredman, M. L., Komlos, J. & Szemeredi, E. “Saving a Sparse Table with O (1) Worst Access Time”. *ACM Journal.* 1984; Vol. 31 No. 3: 538–544.

40. Herlihi, M., Shavit, N. & Tsafrir, M. “Hashing of the classics”. *Proceedings of the 22nd International Symposium on Distributed Computing*. *Publ. Springer-Verlag.* Arcachon: France. 2008. p. 350–364.

Conflicts of Interest: the authors declare no conflict of interest

Received 08.02.2021
Received after revision 26.08.2021
Accepted 23.09.2021

DOI: https://doi.org/10.15276/hait.03.2021.1
УДК 004.65:519.172

Модель і метод подання складних динамічних інформаційних об'єктів на основі LMS-дерев у NoSQL базах даних

Олександр Семенович Максимов[1)]
ORCID: https://orcid.org/0000-0001-8951-5251; oleksandr.s.maksymov@onu.edu.ua
Євгеній Валерійович Малахов[1)]
ORCID: https://orcid.org/ 0000-0002-9314-6062, eugene.malakhov@onu.edu.ua. Scopus: ID: 56905389000
Виталій Іванович Межуєв[2)]
ORCID: https://orcid.org/0000-0002-9335-6131; vitaliy.mezhuyev@fh-joanneum.at. Scopus ID: 24468383200
[1)] Одеський національний університет ім. І.І. Мечникова, вул. Дворянська, 2. Одеса, 65082, Україна
[2)] FH Joanneum: Капфенберг, Werk-VI-Straße 46, 8605, Австрія

АНОТАЦІЯ

У статті виконано аналіз існуючих підходів до опису складних динамічних інформаційних об'єктів при побудові автоматизованих систем управління. Введено і визначено поняття Складного динамічного інформаційного об'єкта. Проведено порівняльний аналіз тимчасових складнощів деревовидних структур і вибраний оптимальний для роботи з

Складним динамічним інформаційним об’єктом. Більшість сучасних автоматизованих систем управління для свого функціонування використовують різноманітні підходи опису об'єктів автоматизації. Під об'єктом автоматизації ми будемо розуміти функціональні об'єкти, які описані у вигляді структурних моделей, що відображають властивості фізичних об'єктів. Таким чином, для оптимальної роботи зі складними динамічними інформаційними об'єктами ми розробили власну модель і метод опису LMS-дерева (Log-structured merge-tree), з можливістю поділу і зберігання до елементарних рівнів. Однією з особливостей нашого підходу до опису об'єктів є наявність деревовидних рівнів – так званих «листів», під якими ми будемо розуміти спеціальні елементи дерева, що розширюють опис деревовидної структури конкретного рівня дерева. Мінімальні елементи листя дерева - «прожилки» - це деталі, тобто елементарні інформаційні елементи. Лист являє собою об'єднання за певними характеристиками «жилок» (деталей), що дають розширену інформацію про рівень об'єкта дерева. Дескриптор елементарного рівня - це множинне поле (масив) NoSQL бази даних, в якому номер рівня дерева є індексом масиву бази даних. Такий підхід дозволяє витягувати і групувати об'єкти відповідно до рівня елементів визначення дерева, що забезпечує швидкий доступ до даних, а також до розширень рівня дерева – «листя».

Ключові слова: Складні; динамічні; інформаційні об'єкти; LMS-дерева; NoSQL; моделі; деревовидні структури

ABOUT THE AUTHORS



Oleksandr S. Maksymov, Senior Lecturer, Department of Mathematical Support of Computer Systems, Odessa I. I. Mechnikov National University, 2, Dvoryanskaya Str. Odessa, 65082, Ukraine
ORCID: https://orcid.org/0000-0001-8951-5251; oleksandr.s.maksymov@onu.edu.ua
***Research field*:** Information technology; business process automation; NoSQL databases; cross-platform systems; storage of unstructured baths; integration systems

Олександр Семенович Максимов, старший викладач, кафедра Математичного забезпечення комп’ютерних систем. Національний університет України «Одеський Національний Університет ім. І. І. Мечнікова», вул. Дворянська, 2. Одеса, 65082, Україна



Eugene V. Malakhov, Dr. Sci. (Eng), Professor of Mathematical Support of Computer Systems Department. Odessa I. I. Mechnikov National University. 2, Dvoryanskaya Str. Odessa, 65082, Ukraine.
ORCID: https://orcid.org/0000-0002-9314-6062; eugene.malakhov@onu.edu.ua. Scopus ID: 56905389000
***Research field*:** Information technology; databases theory; expert systems; system modelling; cloud computing; distributed computing

Євгеній Валерійович Малахов, доктор технічних наук, професор, завідувач кафедри Математичного забезпечення комп’ютерних систем. Національний університет України «Одеський Національний Університет ім. І. І. Мечнікова», вул. Дворянська, 2. Одеса, 65082, Україна



Vitaliy I. Mezhuyev Dr. Sci. (Eng.), Prof., FH Joanneum: Kapfenberg, Werk-VI-Straße 46, 8605, Austria
ORCID: https://orcid.org/0000-0002-9335-6131; vitaliy.mezhuyev@fh-joanneum.at. Scopus ID: 24468383200
Research field: Software engendering; information technology; data analysis

Віталій Іванович Межуєв, д-р технич. наук, професор Інституту промишленного менеджменту. Университет прикладних наук FH JOANNEUM. Капфенберг, Австрія