УДК 004.651.4
DOI: 10.25559/SITITO.18.202201.134-143

Original article

Horizontably Scalable LSM-tree Based Index for Full-Text Search Boolean Queries

A. M. Neganov
Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation
Address: 9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation
neganovalexey@gmail.com

Abstract

Let there be a set of objects, where each object is characterized by Boolean features. Let *logical query* or *Boolean query* denote a query to find objects characterized by a Boolean function by features, for example, “documents with all words ...and any words ...and without words ...”. The terms “object” and “document” are used interchangeably hereinafter. The features may have different semantics, i. e. some features may correspond to the words of the document, some to labels or categories, and some to per-bit time quantum of a document’s date. Although indices executing Boolean queries are well researched in the literature, the common technique of maintaining posting lists is not always acceptable. If the data volume can reach to the order of petabytes, a compact index structure becomes vital. The aim of the research is to suggest a method to build an efficient bitmap index in secondary memory that allows us toupdate or append data being indexed with high write throughput. We propose an efficient LSM-based index for bitmaps and feature design for practical applications. We also discuss aspects of building of joined indices in order to achieve good scalability. The paper describes the architecture and algorithms of a suggested index and include results of our experiments that show sustainable performance of our solution.

Keywords: full-text search, bitmap index, LSM-tree, data-intensive computing, Boolean query

The author declares no conflict of interest.

For citation: Neganov A.M. Horizontably Scalable LSM-tree Based Index for Full-Text Search Boolean Queries. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2022; 18(1):134-143. doi: https://doi.org/10.25559/SITITO.18.202201.134-143

Научная статья

Горизонтально масштабируемый индекс на основе LSM-дерева для логических запросов полнотекстового поиска

А. М. Неганов
ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация
Адрес: 141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9
neganovalexey@gmail.com

Аннотация

Пусть существует набор объектов, где каждый объект характеризуется логическими признаками. Пусть логический запрос или Булевский запрос обозначает запрос на поиск объектов, характеризуемых логической функцией, по признакам, например, "документы со всеми словами... и любыми словами ... и без слов... ". Термины "объект" и "документ" в дальнейшем используются как взаимозаменяемые. Компоненты могут иметь различную семантику, т.е. некоторые компоненты могут соответствовать словам документа, некоторые – меткам или категориям, а некоторые – побитовым квантам времени даты документа. Хотя индексы, использующие логические запросы, хорошо изучены в литературе, общий метод ведения списков рассылки не всегда приемлем. Если объем данных может достигать порядка петабайт, компактная структура индекса становится жизненно важной. Цель исследования – предложить метод построения эффективного растрового индекса во вторичной памяти, позволяющий обновлять или дополнять индексируемые данные с высокой скоростью записи. Мы предлагаем эффективный индекс на основе LSM для растровых изображений и дизайн элементов для практических приложений. Мы также обсуждаем аспекты построения объединенных индексов для достижения хорошей масштабируемости. В статье описываются архитектура и алгоритмы предлагаемого индекса, а также результаты наших экспериментов, которые показывают устойчивую производительность нашего решения.

Ключевые слова: полнотекстовый поиск, битовый индекс, LSM-дерево, вычисления с интенсивным использованием данных, логические запросы

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Неганов А. М. Горизонтально масштабируемый индекс на основе LSM-дерева для логических запросов полнотекстового поиска // Современные информационные технологии и ИТ-образование. 2022. Т. 18, № 1. С. 134-143. doi: https://doi.org/10.25559/SITITO.18.202201.134-143

Introduction

Nowadays both science and industry are affected by explosive growth of data. For a significant amount of time, astrophysics and high-energy physics were famous for their behemoth amounts of data being generated in experiments at daily basis [1], [3], and now biology, especially genomics, uses terabyte-sized datasets and petabyte-sized data repositories, which volume doubles every year, too [4]. The amount of data used and generated in business applications, such as financial and social media analytics, is an another example of exponential growth [5]. In such a circumstances, a feasible data management solution is determined by having abilities to capture and save data into external memory at sustainable rate.

For hard disk drives, the sequential write speed is about 1000 times higher than the random access speed [6]. For solid-state disks, writing in random access mode leads to performance degradation by about an order of magnitude [7, 8], and even despite the fact that the controllers of most modern Flash-media carry out data grouping before writing, sequential write may be about 3 times faster than writing in random access mode [9]. This means that the algorithm must support batching of writes and perform writes to external memory in append-only mode.

Log-structured merge tree (LSM-tree) [10] is a complex multi-level index that solves the problem of performance of writing to external memory by batching changes and writing in sequential manner. An LSM-based index is used by many modern commercial database systems, such as BigTable [11], [12], HBase [13], Cassandra [14], LevelDB [15], RocksDB [16], and AsterixDB [17].

The LSM-tree consists of multiple sorted runs, the size of which grows exponentially, as suggested by the original work [10], or according to another, possibly complex, rules [15, 18]. Updates always go to the input component (usually referred as C_0), which takes residence in the main memory, and, when a sufficient number of them is accumulated, they are sequentially transferred to external memory during so-called merge, or compaction, procedure, that may involve merging of the sorted runs together, including those already written to external memory storage. All sorted runs of an LSM-tree, that are external memory resident, are immutable since they are created during merge procedure. That makes the difference between a LSM-tree and traditional indexes, such as B-tree, that allow in-place updates. Out-of-place update strategy of the LSM-tree eliminates random I/O problem, because all I/Os are sequential. it should be noted that LSM-tree architecture allows concurrent reading while the merge procedure is running, that significantly simplifies concurrency control.

Different components can contain elements with the same keys; in this case, the value in the tree with less number (often called *level*) is taken. Deletion of elements is performed by inserting a special element (*deletion marker*, or *tombstone*) with the required key; the physical removal of an item from the index occurs during the merge procedure that forms the last-level component.

The structure of the components can be varied. Original paper [10] proposes a red-black or AVLtree for the memory component C_0, while most modern implementations adopt some concurrent data structures such as a skip-list or a B+-tree [15]. On-disk components may be organized using B+-trees or so-called sorted string tables (SSTables).

Although LSM-like index is a way to achieve sufficient write performance, it is not clear how to perform complex Boolean queries on such an index[1]. In order to do so, one need to maintain a number of sets such as union, intersections and difference operations can be performed efficiently [2]. Integer indexes are split into blocks of 2^{16} elements in each block in order to make a container for one. Roaring bitmap format allows fast random access, rank and select operations on a container, as well as fast union, intersect and difference operations between containers, even if the containers are of different types [19, 20, 21]. Those operations can be performed on a compressed container without decompression, so if N denotes the length of the block and S the number if significant bits, the operations mentioned above will have $O(S)$ complexity, not $O(N)$, as for uncompressed bitmap.

If the data being indexed is read-only and index is never changed (as in [22], it is obvious to make an index with bitmaps as values corresponding to object features as keys. In order to execute a Boolean query, one should take bitmaps corresponding to the selected features and perform bitwise operations on them. The problem with straightforward approach is following: if the features are the keys of a container, and the bitmaps are its values, then adding a new document to such an index leads to updating a large number of bitmaps, possibly all. therefore a large index in secondary memory on a mutable or expandable data would not be functional. In this paper, we study how to overcome this difficulty and make an LSM-based index with bitmaps that allows to add, update and remove data efficiently. Then, we discuss how to scale our solution up.

The paper is organized as follows. In the next section, we present an overview of our architecture. Next, in Section 4 we discuss dictionary design in the view of overall scalability. Then, the experimental results are shown in Section 5. Finally, we conclude with Section 6.

Design and Implementation

1.1 Search Tree

We assume that the objects (documents) to be indexed are stored in some key-value storage and each object can be retrieved by a unique key. Hence the bitmap index being described is a secondary index that allows to obtain an object's key by its features[2].

From the logical point of view, we assign a bitmap for each feature, so we can execute Boolean queries in a straightforward way. In example, in order to obtain a set of objects that have both feature X and feature Y, we can perform bitmap intersection for corresponding features, that can be done efficiently, as stated before [20]. However, we do not store a huge bitset as is. Following the approach suggested in [19], we split a logical bitmap into blocks of a certain size, less than or equal to 2^{16}. Apart from better compression ratio, it is used to return paged query output (with page start token and size) efficiently, so the result page of given size starting from the given offset can be computed using a limited amount of blocks, not all logical bitset. In order to do so, bitmap blocks are stored as individual entities and can be retrieved and processed independently.

[1] Manning C.D., Raghavan P., Schütze H. Introduction to Information Retrieval. USA: Cambridge University Press; 2008. 506 p. (In Eng.)

[2] Pilosa: A Technical Overview. Tech. rep. Dec. 2017. 7 p. [Electronic resource]. URL: https://www.pilosa.com/pdf/PILOSA%20-%20Technical%20White%20Paper.pdf (accessed 13.01.2022). (In Eng.)

	block 1				block 2	
	doc 1	doc 2	doc 3	doc 4	doc 5	...
feature 1	0	0	1	0	1	...
feature 2	1	1	0	0	1	...

F i g. 1. An overview of bitmap index with blocks

For each object feature (i. e. word / label / date) an ID is assigned; on the matter of feature to feature ID mapping, see Section 4. For each document, a number *document ID* is assigned to it, that determines the document unambiguously. That document ID is to be translated into index in a bitmap with formula

$$\text{document ID} = \text{block ID} \times \text{block size} + \text{integer in a block bitmap} \quad (3.1)$$

A document ID is to be assigned as a per-index incremental counter. The advantage of this approach is that newer documents (objects) will have greater document IDs, so we can perform paginated queries in the order of the document's index time.

The bitmap blocks are to be stored in the LSM-tree (referred as *search tree* hereinafter), where keys corresponds to *(block ID, feature ID)* pair, and values are compressed bitmap blocks.

Since the document ID to assign always increments, the current block ID always increments too. Therefore insertions of the nodes with the keys of the form *(block ID, feature ID)* are partially ordered in key order. It implies that the merging of the LSM sorted runs may be trivial since the key ranges of the runs likely do not intersect, and merging can even be omitted since the insertions in the block order results in the sorted runs that can be viewed as partitions in a partitioned LSM [15], that means much lesser write amplification. Moreover, if the older documents is considered historical and are to be accessed infrequently, the sorted runs / partitions might be stored on a cheaper cold storage [23-25] or on the write-once-read-many (WORM) devices.

In order to obtain a complete logical bitmap for a feature with the ID equal to X, one should get the blocks $(b_1,X),(b_2,X),\ldots,(b_{nX},X)$, where $b_1,b_2,\ldots,b_{nX}$, $0 \le b_1 < b_2 < \cdots < b_{nX}$ is the sequence of the block IDs such that for each i, $0 \le i \le n_X$ a document with the number within a block with the feature X exists (the blocks of zeroes should not be stored) and for each i such that $i > n_X$ there is no blocks with the ID equal to i for the feature X. Hence in order to get IDs of the documents that have feature X, one shoud perform n_X+1 "greater or equal" lookups in a tree, from the key $(0,X)$ to the $(b_{nX}+1,X)$, and enumerate the significant bits in the blocks. The complexity of the operation is $O(n_X \log F)$, where F is the total number of features.

In order to get ID of the documents that have both feature X and feature Y, one should perform an intersection operation on the corresponding bitmap blocks. In general, in order to obtain documents that are characterized by some Boolean function on features f, one should perform queries for corresponding features and execute f on the blocks. It is always possible, as any Boolean function f may be converted to disjunctive normal form or conjunctive normal form, so it can be calculated using union, intersection and difference operations on bitmaps.

The compound key defined would allow to paginate search results, since on the first request only N blocks may be processed, the ID of N-th block and an index within that block may be encoded as a page token, on second request blocks from N to $N + M$ may be processed, and so on.

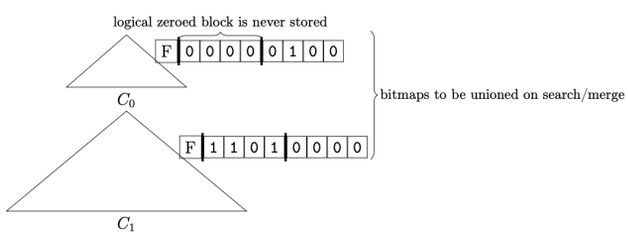
The insertion into index is a bit tricky. As was mentioned above, we do not want to update a lot of bitmaps / bitmap blocks when on a new document indexing procedure. Instead, we want to update only bitmaps in the memory component C_0; even lookup in on-disk components of the LSM-tree should not be performed.

The document ingestion algorithm is following.

Algorithm 3.1 Insert a new document into a search tree

lastDocId ⇐ a value from global head *docId* ⇐ *lastDocId* + 1
global head counter ⇐ *docId*
blockId ⇐ *docId*/*blockSize*
idx ⇐ *docId* mod *blockSize*
for all *feature* ∈ document features do
insert a bitmap of a single significant bit *idx* to an LSM
end for

Unlike usual LSM-tree implementations, an older value is not to be replaced by a newer one during LSM merge procedure. Instead, a union of bitmap values (bitwise OR) should be performed. Lookup procedure in the search tree does not stop on the first match, as in regular LSM-tree. Instead, it finds results from all components and performs union operation on them. Therefore Bloom filtering is important to skip some components from the search.



F i g. 2. A search tree with feature F

1.2 Overall Architecture

In order to translate document ID, that is calculated by formula 3.1, to a primary key-value storage key, a special tree is needed, referred as *mapping* hereinafter. Mapping is a regular LSM-tree where document IDs serves as keys and primary index keys as values.

In order to convert text words, physical quantity value, etc. to feature ID, a special dictionary tree may be needed. The matter is discussed in Section 4 below.

In a regular mode of operation, documents are never removed from search tree, and last document ID counter never decrements. In order to filter deleted documents out, a special deleted documents filter is needed. That is a LSM-tree with block IDs as keys and bitmaps of deleted documents as values. To filter deleted documents out, one need to calculate the difference between bitmap calculating using search tree (see 3.3 below) and the bitmap of the corresponding block retrieved from deleted documents filter. If there's B blocks in

the index and F features, the size of the search tree is proportional to $B \cdot F$ in the worst case, while the size of the deleted documents tree is proportional to just B. The problem of garbage collection is discussed in Section 3.4.

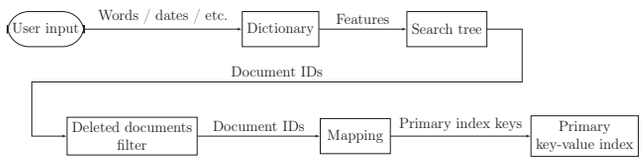


Fig. 3. Search request flow

1.3 Nested Iterators

In a practical application, per-feature bitmaps may be very sparse, so many logical blocks may contain no significant bits and therefore be not present in a search tree at all. In such conditions, it would be very impractical to traverse LSM block-by-block and compute unions or intersections of blocks with empty ones. Instead, computations should be performed only when non-empty result is possible. For example, the logical OR (union) of bitmap blocks may be not empty only if there's at least some non-empty block with that block ID, in other words, a block stored in the tree. Then, to the logical AND (intersection) of bitmap blocks may be not empty only if all blocks with the block ID are stored. That means that if there is a feature F_1 that a lot of documents have, and a feature F_2 that a few documents have, a lot of bitmap blocks that are stored for feature F_1 may be skipped from computation.

In order to achieve this an iterator can be implemented, that takes some block ID as start, finds out first block ID with non-empty blocks and computes the result for them, returning this block ID along with the computed bitmap. Furthermore, an iterator can be built on top of another iterators. A Boolean query can be represented as a tree with features as leaf nodes and logical operators in directory nodes, for example, AND, OR, NOR, NOT. A skipping iterator can be built for each node, with the topmost iterator returning results of the whole query. This approach allows executing even complex queries efficiently.

Algorithm 3.2 Pseudo-code implementation of AND iterator

```
interface BlockIterator {
        NextAfter() (blockID BlockID, result *Bitmap, empty Boolean, err Error)
};
struct andIter { iters
        []BlockIterator
};
func (ai *andIter) NextAfter(start BlockID) (BlockID, *Bitmap, Boolean, Error) {
        var results = new([ai.iters.length]*Bitmap);
        BEGIN:
                for it in ai.iters {
                        position, bmap, empty, err := it.NextAfter(start);
                        if err != NULL or empty {
                                // x AND 0 = 0, no results at all
                                return 0, NULL, true, err;
                        }
                        if position > start {
                                start =
                                position; goto
                                BEGIN;
                        }
                        results.append(bmap);
                }
        return nextBlock, results.And(), false, NULL;
}
```

1.4 Document deletion

In order to support deletion of the documents, the special tree that maps the block ID to the bitmap of the removed documents is added. It allows to mark the document as deleted quickly and then filter deleted documents out calculating the difference between resulting bitmap from the search tree and the deleted documents bitmap.

As the number of deleted documents grows, it slows down index lookups due to processing of garbage to be filtered. Moreover, if the objects being indexed are mutable, each new object version will correspond to a new "document", so the amount of garbage for a frequently changing data may increase quickly.

The idea of garbage collection procedure follows. Let a threshold be the percent of removed documents per block, for example, 1/2 or 2/3. A block is subject to garbage collection if the number of removed documents within a block exceeds the threshold. A bitmap of the blocks that need garbage collection should be maintained, i. e. it is possible to update that bitmap when a document is deleted (with obvious optimizations). The procedure itself starts if the bitmap of block that need garbage collection is not empty. It traverses all keys of the search tree then, i. e. all *(feature ID, block ID)* pairs.

For each pair, if the difference between bitmap for this feature and block and deleted documents bitmap for the block is empty, the *(feature ID, block ID)* key can be removed from the search tree (by deleted marker insertion). If it is done for all features of the block, deleted documents filter can be cleared too. Otherwise, deleted documents bitmaps remain, but "needs GC" bit is cleared until a new document from this block is removed.
The advantage of the algorithm above is that it does not require significant memory for execution and does not break "greater document IDs for newer documents" rule.

Dictionary and consolidated index

1.1 Word Features

The most important document feature kind for a full-text search engine is features of having a specific word in a document, called *word features* hereinafter. In order to support such features, each word should be mapped to a feature ID, that is to be used in a search index key. There is a number of choices on how to do it. First of all, the word token can be used directly (after the usual analysis and tokenization), the second option is hashing and the final one is to add an another tree.
The primary disadvantage of the word tokens as the feature IDs is their arbitrary length, so they would take a lot of space in the search index. Hashing can partially resolve this problem, however, hash values should be large enough to avoid collisions, they are random and hard to compress. Moreover, it is impossible to find minor variants of the term in the index, because it can be hashed to a very different hash value, or to enumerate all words in an index, which may be useful for analytical purposes.
A special search tree for a dictionary overcome those issues. It maps a word token into an integer, that is used as an ID in the search index. These IDs are relatively small and easy to compress due to their incremental nature. In order to achieve high write throughput, a LSM-tree should be used.
In order to search by prefix, one can make a range query in a dictionary tree, get a list of features in a range, and then make an OR query on those features in a search tree with document bitmaps.
If the set of words is more or less stable, for example, if the words are from some natural language, keys may be never deleted from the dictionary tree, like in Postgres GIN indexes. Otherwise, in order to perform a garbage collection procedure, one should traverse through the whole dictionary tree and for each feature lookup for a corresponding element in a main search tree; the elements with no corresponding bitmap blocks found are to be removed.

1.2 Datetime Features

It is suggested to adopt Pilosa's bit-sliced indexing (BSI) technique [1], with respect to Gregorian calendar (as it is likely that the user will choose desired time range in a calendar form). For example, a year between 1900 and 2155 can be represented as collection of 8 Boolean features: 1st would be 0 if the year is between 1900 and 2027 and 1 if it is between 2028 and 2155, 2nd is 0 for 1900–1963 / 2028–2092 and 1 for 1964–2027 / 2092–2155 and so on. Therefore time range 2020–2025 can be encoded as "NOT 1st 1900–2027 AND 2nd 1964–2027 AND 3rd 1996–2027 AND 4th 2012–2027 AND 5th 2020–2027 AND (NOT 6th 2020–2023 OR (6th 2024–2027 AND NOT 7th 2024–2025)). Then, we can use 4 features/bitmaps for a month, 5 for a day and so on.
In order to reduce the number of bitmaps involved, a special feature can be used for each calendar year, and bit-sliced indexing can be used for a date-time within a year. For example, we can make a feature ID in a following way: feature type magic + year code for year features or the order of the binary division for in-year features. For each actual datetime "feature" of a document (for example, document creation timestamp) there would be a number of the search index features: first, a year feature; second, the "first division" feature, if the date is in the second half of the year; third, the "second division" feature, if the date is in the second or fourth quarter of the year, and so forth, and so on.
It is worthy of note that a dictionary index is not needed to work with datetime features, their IDs is to be constructed on the fly.

1.3 Consolidated Index

While each database can benefit from more powerful hardware, true scalability comes from the ability to add more nodes to the cluster and to spread load between them. Read scalability can be achieved through replicas that receive copied data from the primary node. It is less straightforward how to make a horizontally scalable system in terms of write operations.
The possible solution is to have a number of independent primary nodes with a search index, that is the search tree, dictionary tree and mapping tree as described above, and a special common dictionary, that allows to omit searching in all nodes. That common dictionary should comprise a mapping from the word token to the list of nodes that have a document with this word. Said list is supposed to be stored as a compressed bitmap.
The single word query execution is simple, one just need to select relevant nodes from the corresponding list and make search queries on this nodes. In order to execute more complicated queries, one may need to calculate unions, intersections and differences of such lists. Of course if a node has a document with the word *A* and it has a document with the word *B*, it is not necessary that it has a document with both *A* and *B*. However, the common dictionary still significantly restricts the set of the nodes to be checked. Datetime queries can be executed either purely on the second stage, when a search on the node index is performed, or can be accelerated with the common dictionary too. In order to do this, one need to choose a "primary" date field of the documents (for example, creation or modification date) and construct a common dictionary key as a word + year number (or as a word + year + month, or even more granular, dependent on the application). It may significantly improve performance of "documents with words and within a time range" queries in a distributed index, if they are important.
In order to maintain such a common dictionary, each node can maintain a log of the new words or word – year pairs that are indexed since the first synchronization. The common dictionary then can be updated with a background procedure that polls the nodes and updates the dictionary with changes it got. If the vocabulary size is within reasonable limits, the common dictionary updates are not a bottleneck, especially considering that the new elements can be inserted into nodes indexes concurrently not waiting for a common dictionary update.

Evaluation

1.4 Experimental Setup

Implementaion

LSM tree is written in Go code and consists of approximately 5,500 lines of code. It relies on an abstract I/O layer that encapsulates actual storage device and reports various metrics with approximately 1,000 lines of code. Search index is implented with approximately 5,000 lines of code.

The LSM implementation uses simple levelled compaction policy [15]. It is generally suboptimal in terms of write amplification and insert performance; however, there is a known optimization for a key-order insert workload that reduces write amplification a lot. The major advantages of the levelled policy is that it implies relatively low space amplification and fewer number of components, therefore it is characterized by better read performance than other policies.

LSM data is compressed with Zstandard algorithm prior to be written to external memory device. Point lookup queries are optimized with Bloom filters.

For a mapping tree, the primary storage keys are 16 bytes long.

Experiment machine

The experiment machine is an Intel Core i5 (4 cores at 2.9 GHz) with 8 GB of RAM and a 1 TB SSD drive.

Dataset. The experiments in this section are performed with a CommonCrawl data. The CommonCrawl is an open repository of web crawl data. Said data is an enormous collection of text written primarily by humans in natural languages, therefore the distribution of the word features may be close to one of the real documents in information retrieval system. The vocabulary of the CommonCrawl text is plenty rich due to nearly all human languages present in the global web crawl data, and the documents may contain a vast amount of some identifiers and other random byte sequences, so the dictionary size is not limited in any way. Text data in WET format are split into documents to be indexed of 50 KiB each. For each document, a date between 1 January 1980 and 1 January 2050 is assigned using an uniform random distribution.

Defaults

The threshold for the C_0 tree was set to 8 MiB, so for each of the three LSM trees involved (search tree, dictionary tree, mapping tree) the maximum size for in-memory component is 8 MiB. The block size of 2048 bits was chosen for the search tree.

1.5 Evaluation Results

In Figure 4, we study the impact of the actual amount of bytes written to external memory device

$S_{written}$ on the total size of documents inserted S_{docs}. The results show that the write amplification ratio has no trend to increase.

In Figure 5, the dependency of the average insertion time τ_i on the index size (in the number of documents of 50 KiB each N_{docs}) is illustrated. In this experiment, the time to insert each 10,000 documents is measured and the average is calculated by deletion by 10,000. The results are coherent with the results shown on Figure 4 and show that the insertion time has no trend to increase with the growth of the index.

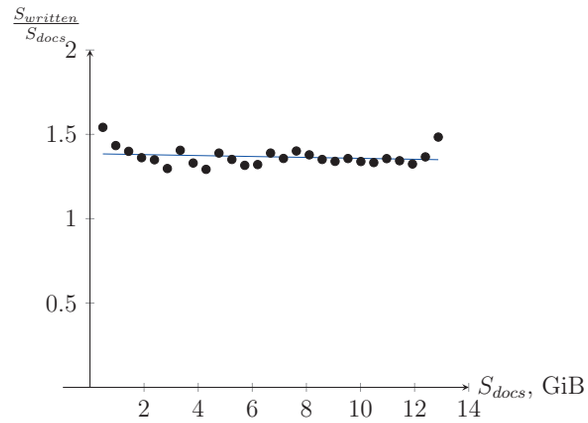


Fig. 4. Impact of index size on write amplification

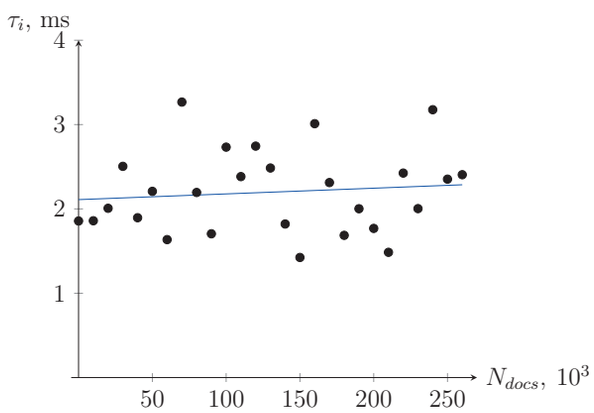
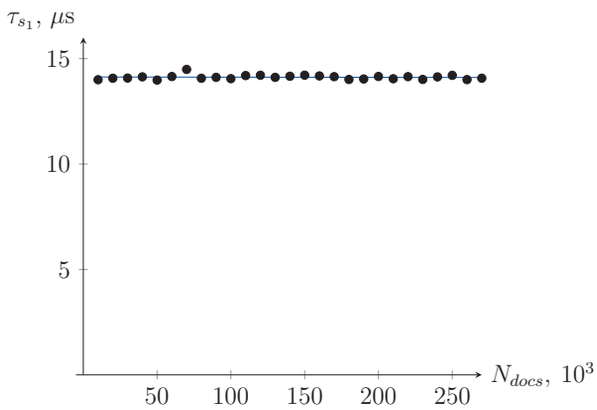
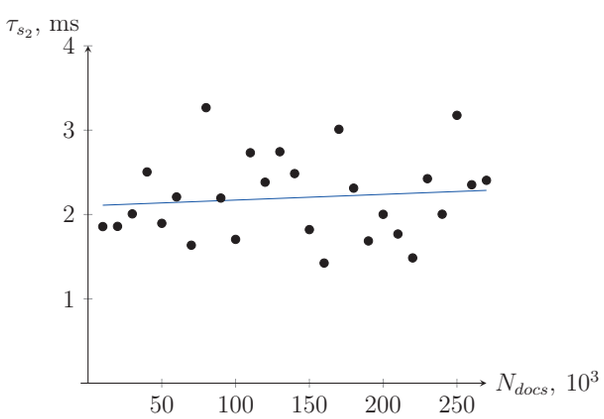


Fig. 5. Impact of index size on insert time

In the next several sets of experiments, the impact of the index size on the search time is studied. The workload issues insert and read operations. After each 10,000 documents inserted, the average search time is measured for the "simple" and "complex" query. The "simple" one is for the documents containing the English word *is*; since the vast majority of the documents contains it, the query should be answered relatively quickly. The "complex" one can be expressed as follows: "document, containing *search* and *success* and either *accomplished* or *achieved*, and neither *problem*, nor *bug* in date range from 5th March 1990 to 16th November 2016". The results illustrated in Figure 6 and Figure 7 show that the time to answer the "simple" query τ_{s1} is significantly inferior to the time to answer the "complex" query τ_{s2}, while both show no trend to increase with the index size.

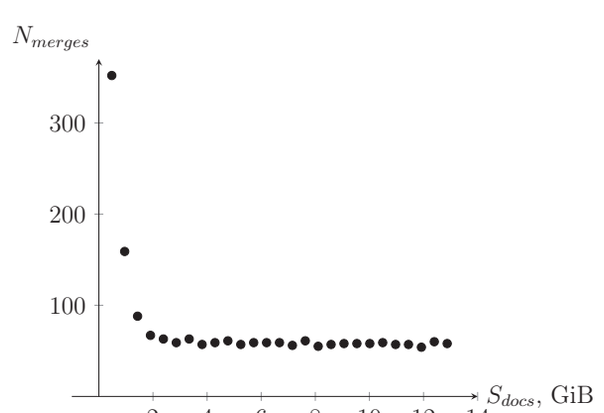


F i g. 6. Impact of index size on “simple” search time



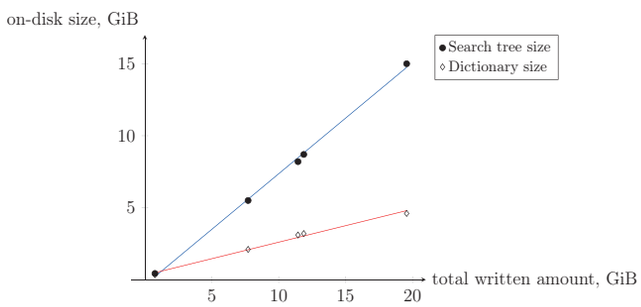
F i g. 7. Impact of index size on “complex” search time

In Figure 8 we study the amount of LSM merge operations required to insert 10,000 documents
N_{merges} on the index size before the insertion S_{docs}. The results show that the number of merges is stable, with a initial peak.



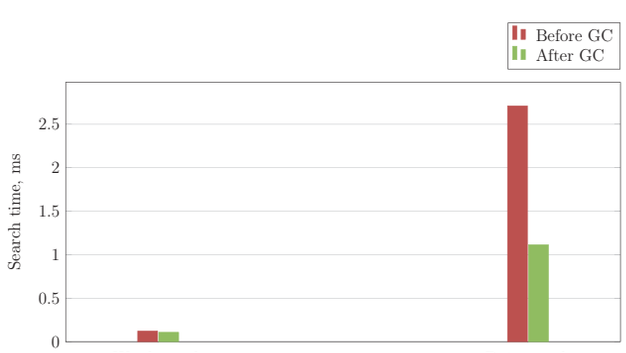
F i g. 8. Impact of index size on total number of LSM merges to insert 104 documents

In Figure 9 we compare the sizes of the main search tree with the bitmaps and the dictionary tree. The results show that although the dictionary size grows linearly (because of the vocabulary-rich dataset), the search tree size grows much quicker. It should be noted that the mapping tree size is limited to hundreds of kilobytes.



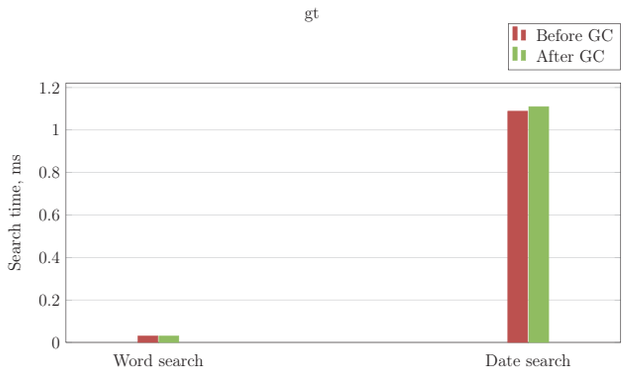
F i g. 9. Sizes of the search tree and the dictionary

The impact of the suggested garbage collection procedure for the search time is analyzed in the following experiments. In Figure 10 we study the impact of garbage collection on the search time, where the index comprises 8,000 documents, 2/3 of which were deleted, and the deleted and remaining documents are mixed uniformly, so for all block number there may be ones that need garbage collection. Two search queries were evaluated. The “simple” one is for the documents included a word *search*, and the “complex” one is for the documents with creation date between 5 March 2005 and 16 November 2006, so that many features are involved, according to the date encoding.



F i g. 10. Impact of the garbage collection on the search time in case of uniformly distributed deletions

The results show that the suggested garbage collection procedure decreases the search time a lot for the complex queries, for example, for a date query, that involves querying for a lot of features. The “simple” query, however, seems to execute fast enough with deleted documents filter.
In Figure 11 we study the garbage collection on the index where 8,000 documents were inserted and then the oldest 2/3 were deleted, so the blocks subject to the garbage are consequent, and the search is performed in “from the newest to the oldest” order.



F i g. 11. Impact of the garbage collection on the search time in case of the oldest documents deleted

The results unsurprisingly show that the garbage collection does not improve the search time in the case, because there were no documents to filter out. However, it may be still needed to decrease the stored index size.

Conclusions and future work

A horizontally scalable bitmap search index with out-of-place writes and ability to execute arbitrary complex Boolean queries have been designed and implemented. The possible applications of the method include, but not limited to a full-text search engine that implements searching for the text documents by words contained and timestamps. However, the method being described can utilise a completely different kind of a document features, with possible changes in dictionary design.

The index implementation achieves high performance, both in terms of write throughput and search time, that proves the design choices. However, the implementation evaluated can be viewed as a first version of the product and there are ways to further enhance its performance. In particular, the compaction policy may be tuned to decrease write amplification.

References

[1] Foster I., Kesselman C., Tuecke S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of High Performance Computing Applications.* 2001; 15(3):200-222. (In Eng.) doi: https://doi.org/10.1177/109434200101500302

[2] Ponte J.M., Croft W.B. A language modeling approach to information retrieval. *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'98).* Association for Computing Machinery, New York, NY, USA; 1998. p. 275-281. (In Eng.) doi: https://doi.org/10.1145/290941.291008

[3] Mattmann C.A. A vision for data science. *Nature.* 2013; 493(7433):473-475. (In Eng.) doi: https://doi.org/10.1038/493473a

[4] Marx V. The big challenges of big data. *Nature.* 2013; 498(7453):255-260. (In Eng.) doi: https://doi.org/10.1038/498255a

[5] Chen C.L.P., Zhang C.-Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences.* 2014; 275:314-347. (In Eng.) doi: https://doi.org/10.1016/j.ins.2014.01.015

[6] Anderson D., Dykes J., Riedel E. More Than an Interface – SCSI vs. ATA. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03).* San Francisco, CA: USENIX Association; 2003. p. 245-257. Available at: https://www.usenix.org/conference/fast-03/more-interface%E2%80%94scsi-vs-ata (accessed 13.01.2022). (In Eng.)

[7] Chen F., Koufaty D.A., Zhang X. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. *ACM SIGMETRICS Performance Evaluation Review.* 2009; 37(1):181-192. (In Eng.) doi: https://doi.org/10.1145/2492101.1555371

[8] Picoli I.L., et al. UFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives. *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17).* Association for Computing Machinery, New York, NY, USA; 2017. Article number: 20. p. 1-7. (In Eng.) doi: https://doi.org/10.1145/3124680.3124741

[9] Hu X.-Y., et al. Write Amplification Analysis in Flash-based Solid State Drives. *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR'09).* Association for Computing Machinery, New York, NY, USA; 2009. Article number: 10. p. 1-9. (In Eng.) doi: https://doi.org/10.1145/1534530.1534544

[10] O'Neil P., et al. The Log-structured Merge-tree (LSM-tree). *Acta Informatica.* 1996; 33(4):351-385. (In Eng.) doi: https://doi.org/10.1007/s002360050048

[11] Chang F., et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*. 2008; 26(2):4. (In Eng.) doi: https://doi.org/10.1145/1365815.1365816

[12] DeCandia G., et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*. 2007; 41(6):205-220. (In Eng.) doi: https://doi.org/10.1145/1323293.1294281

[13] Hassan M.U., et al. A Comprehensive Study of HBase Storage Architecture – A Systematic Literature Review. *Symmetry.* 2021; 13(1):109. (In Eng.) doi: https://doi.org/10.3390/sym13010109

[14] Lakshman A., Malik P. Cassandra – A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review.* 2010; 44(2):35-40. (In Eng.) doi: https://doi.org/10.1145/1773912.1773922

[15] Luo C., Carey M.J. LSM-based storage techniques: a survey. *The VLDB Journal.* 2020; 29(1):393-418. (In Eng.) doi: https://doi.org/10.1007/s00778-019-00555-y

[16] Dong S., et al. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Transactions on Storage*. 2021; 17(4):26. (In Eng.) doi: https://doi.org/10.1145/3483840

[17] Alsubaiee S., et al. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment.* 2014; 7(14):1905-1916. Available at: https://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf (accessed 13.01.2022). (In Eng.)

[18] Lim H., Andersen D.G., Kaminsky M. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16).* Santa Clara, CA: USENIX Association; 2016. p. 149-166. Available at: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lim.pdf (accessed 13.01.2022). (In Eng.)

[19] Chambi S., et al. Better Bitmap Performance with Roaring Bitmaps. *Software: Practice and Experience.* 2016; 46(5):709-719. (In Eng.) doi: https://doi.org/10.1002/spe.2325

[20] Lemire D., Ssi-Yan-Kai G., Kaser O. Consistently faster and smaller compressed bitmaps with Roaring". *Software: Practice and Experience.* 2016; 46(11):1547-1569. (In Eng.) doi: https://doi.org/10.1002/spe.2402

[21] Wang J., et al. An Experimental Study of Bitmap Compression vs. Inverted List Compression. *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17*). Association for Computing Machinery, Chicago, Illinois, USA; 2017. p. 993-1008. (In Eng.) doi: https://doi.org/10.1145/3035918.3064007

[22] Pinar A., Tao T., Ferhatosmanoglu H. Compressing Bitmap Indices by Data Reorganization. *Proceedings of the 21st International Conference on Data Engineering (ICDE'05).* USA: IEEE Computer Society; 2005. p. 310-321. (In Eng.) doi: https://doi.org/10.1109/ICDE.2005.35

[23] Hsu Y.-F., et al. A Novel Automated Cloud Storage Tiering System through Hot-Cold Data Classification. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE Press, San Francisco, CA, USA; 2018. p. 492-499. (In Eng.) doi: https://doi.org/10.1109/CLOUD.2018.00069

[24] Dayarathna M., Wen Y., Fan R. Data Center Energy Consumption Modeling: A Survey. *IEEE Communications Surveys & Tutorials*. 2016; 18(1):732-794. (In Eng.) doi: https://doi.org/10.1109/COMST.2015.2481183

[25] Liu S., Huang X., Fu H., Yang G. Understanding Data Characteristics and Access Patterns in a Cloud Storage System. *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. Delft, Netherlands; 2013. p. 327-334. (In Eng.) doi: https://doi.org/10.1109/CCGrid.2013.11

Submitted 13.01.2022; approved after reviewing 05.03.2022; accepted for publication 16.03.2022.
Поступила 13.01.2022; одобрена после рецензирования 05.03.2022; принята к публикации 16.03.2022.

About the author:

Aleksei M. Neganov, Postgraduate Student, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0000-0003-4451-5332,** neganovalexey@gmail.com

The author has read and approved the final manuscript.

Об авторе:

Неганов Алексей Михайлович, аспирант, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0000-0003-4451-5332,** neganovalexey@gmail.com

Автор прочитал и одобрил окончательный вариант рукописи.