Pamukkale Univ Muh Bilim Derg, 28(5), 720-729, 2022

Pamukkale Üniversitesi Mühendislik Bilimleri Dergisi

Pamukkale University Journal of Engineering Sciences

A method to improve full-text search performance of MongoDB

MongoDB'nin tam metin arama performansını iyileştirme yöntemi

Altan MESUT[1*], *Emir ÖZTÜRK*[1]

[1]Department of Computer Engineering, Engineering Faculty, Trakya University, Edirne, Turkey.
altanmesut@trakya.edu.tr, emirozturk@trakya.edu.tr

Received/Geliş Tarihi: 20.07.2021 Revision/Düzeltme Tarihi: 11.11.2021 doi: 10.5505/pajes.2021.89590
Accepted/Kabul Tarihi: 10.12.2021 Research Article/Araştırma Makalesi

Abstract

B-Tree based text indexes used in MongoDB are slow compared to different structures such as inverted indexes. In this study, it has been shown that the full-text search speed can be increased significantly by indexing a structure in which each different word in the text is included only once. The Multi-Stream Word-Based Compression Algorithm (MWCA), developed in our previous work, stores word dictionaries and data in different streams. While adding the documents to a MongoDB collection, they were encoded with MWCA and separated into six different streams. Each stream was stored in a different field, and three of them containing unique words were used when creating a text index. In this way, the index could be created in a shorter time and took up less space. It was also seen that Snappy and Zlib block compression methods used by MongoDB reached higher compression ratios on data encoded with MWCA. Search tests on text indexes created on collections using different compression options shows that our method provides 19 to 146 times speed increase and 34% to 40% less memory usage. Tests on regex searches that do not use the text index also shows that the MWCA model provides 7 to 13 times speed increase and 29% to 34% less memory usage.

Keywords: NoSQL, MongoDB, Text index, Full-Text search, MWCA.

Öz

MongoDB'de kullanılan B-Tree tabanlı metin dizinleri, ters çevrilmiş dizinler gibi farklı yapılara kıyasla yavaştır. Bu çalışmada, metindeki her farklı kelimenin yalnızca bir kez yer aldığı bir yapı indekslenerek tam metin arama hızının önemli ölçüde artırılabileceği gösterilmiştir. Daha önceki çalışmalarımızda geliştirilen Çok Akışlı Kelime Tabanlı Sıkıştırma Algoritması (MWCA), kelime sözlüklerini ve verileri farklı akışlarda saklar. Belgeler bir MongoDB koleksiyonuna eklenirken MWCA ile kodlanmış ve altı farklı akışa ayrılmıştır. Her akış farklı bir alan ismi ile saklanmış ve bunlardan benzersiz kelimeler içeren üçü metin dizini oluşturulurken kullanılmıştır. Bu sayede indeks daha kısa sürede oluşturulabilmiş ve daha az yer kaplamıştır. MongoDB'de kullanılan Snappy ve Zlib blok sıkıştırma yöntemlerinin MWCA ile kodlanan veriler üzerinde daha yüksek sıkıştırma oranlarına ulaştığı da görülmüştür. Farklı yöntemler ile sıkıştırılan koleksiyonlar üzerinde oluşturulan metin dizinlerinde yapılan arama testleri, yöntemimizin 19 ila 146 kat hız artışı ve %34 ila %40 daha az bellek kullanımı sağladığını göstermiştir. Metin dizinini kullanmayan regex aramaları ile ilgili testler de MWCA modelinin 7 ila 13 kat hız artışı ve %29 ila %34 daha az bellek kullanımı sağladığını göstermiştir.

Anahtar kelimeler: NoSQL, MongoDB, Metin dizini, Tam metin arama, MWCA.

1 Introduction

NoSQL databases are generally used for storing big data. Data compression methods are used to store such large amounts of data on fewer disks/nodes. Since there is a large amount of data to be added every second, compression needs to be fast so that data can be recorded instantly. In order not to slow down the reading speed, a fast decompression speed is also expected. Many NoSQL databases such as MongoDB, Cassandra, Couchbase, LevelDB store their data using the Snappy [1] compression algorithm, which performs very fast compression and decompression. The Zlib [2] algorithm, which compresses data slower but has a better compression ratio, can also be used in MongoDB. The Zstd [3] algorithm, which has a speed close to Snappy and offers similar compression ratios as Zlib, has also been added to compression options in MongoDB version 4.2. Prefix compression is preferred for compressing indexes, where queries can run directly on the compressed index without decompressing.

NoSQL databases, which can return results faster when searching through indexed keys, search slowly on other fields due to their schema-independent structure. For example, a query that finds which of the documents contains a given word (similar to LIKE in SQL) runs slowly on a document-based database such as MongoDB. If a text index is created in the collection and the same search is made in that index, the search time will be significantly shortened. However, since the indexes need to be updated when new documents are added to the database, the delay in insertion increases with the number of documents inserted.

When a text index is created by selecting a specific language, stop words filtering and stemming operations are performed, so that the index uses less space and queries run faster. As of today, MongoDB can create text indexes specific to 15 different natural languages (English is the default language). Therefore, when a query such as LIKE type or Regex type is carried out on English documents, words such as 'the', 'and', 'of' can be found, but not when the query is made through the text index.

MWCA [4], a word-based compression algorithm that we developed in a previous study, sorts all words according to their frequencies, adds the most frequent 255 words to the D1 dictionary and encodes them as 1 byte and the next 65536 words to the D2 dictionary and encodes them as 2 bytes. Although there have been many different studies in the field of text compression in recent years [5]–[9], the fact that MWCA stores dictionaries and data in different streams provides an important advantage for this study in which only dictionaries

*Corresponding author/Yazışılan Yazar

are indexed. The advantages of indexing only the word dictionaries created by MWCA instead of indexing the entire documents will be explained in the fourth section.

There are some studies in the literature on the improvement of MongoDB's full-text search performance. Morishima and Matsutani used the power of the GPU to speed up regex searches in MongoDB that do not need indexes [10]. They proposed a cache mechanism called DDB Cache (Document-oriented DataBase Cache) to accelerate such queries by using GPUs. To perform full-text search on the files stored on GridFS, Kelec et al. transferred the contents of the files to MongoDB documents, created a text index on the related collection and found which files contained the given text by searching through the text index [11]. Truică and Boicea proposed some methods to build an inverted index for documents stored in MongoDB and Oracle databases [12]. Some of these methods take advantage of the frameworks and tools provided by the database systems to build the index, like MapReduce framework for MongoDB and Pipelined Table Functions for Oracle. It was shown in the test results of the article that using the suggested inverted index structure instead of MongoDB's text index structure makes text search queries up to 8 times faster.

The inverted index structure [13] is used also in the Elasticsearch search engine, where more complex text queries can be performed compared to MongoDB text indexes. Elasticsearch can also be used as a database that stores documents. However, it is generally not preferred as a primary database due to the lack of transactions, poor user management, and slow add / update operations. Greca et al. presented an application that uses MongoDB to store agricultural data and Elasticsearch to search data [14]. In Han and Zhu's study, media data was first stored on MongoDB, then the data was transferred to an Elasticsearch cluster and the search speed was increased with the created inverted index [15]. In the study of Lu et al., the text data obtained from the internet was stored in MongoDB, and the performance difference was shown by searching with both MongoDB and ElasticSearch [16] Atlas Search, one of the new features of Mongo DB Atlas (MongoDB's DBaaS platform), uses the Apache Lucene library, which is also used by Elasticsearch. With the help of the additional features of the Lucene library and the inverted index, Atlas Search can perform much faster and more advanced searches than MongoDB text index [17]. Although the method proposed in this study does not provide additional search features, it does provide more accurate search results (discussed in Section 4.2) and MongoDB text indexes to run at inverted index speed.

This paper is organized as follows: Information about the storage and indexing structures in MongoDB is given in the second section. Brief information about the MWCA compression algorithm used to increase search performance on indexes is given in the third section. The advantages obtained when indexing on MWCA model are presented with experimental results in the fourth section. The obtained results were evaluated in the last section.

2 MongoDB

MongoDB is the most preferred document-based database today. It is an open-source project developed with C++. It started to be developed by 10gen in 2007. The company later changed its name to "Mongodb Inc.". MongoDB has a unique query language and is easy to scale horizontally. It has started to close its shortcomings according to relational databases by supporting joins in queries with version 3.2 and distributed multi-document ACID transactions with snapshot isolation with version 4.0. It can be used in Windows, MacOS, Solaris and Linux distributions. The write speed of MongoDB is better than the write speed of relational databases. This feature provides advantages in scenarios such as storing large-scale IoT data [18] and being used in the ETL layer of Real-Time Data Warehousing [19].

The document-based database can be defined as a subclass of the key-value databases. The data stored as a 'document' in document-based databases consists of certain fields and can be queried over these fields, while no meta-data is kept in the database about the data stored as 'value' in key-value databases. In document-based databases, data is usually stored in a structured format such as XML or JSON. Results in [20] indicate that JSON is faster and uses fewer resources than XML.

The structure containing the documents in MongoDB is called 'collection'. This structure can be thought of as a 'table' that stores records in relational databases. The biggest difference between them is that the tables have a certain schema; that is, all the records they contain must have the same fields in the predefined data types. The documents in the collections are schema independent; that is, each can have different number of fields with different data types and different names. For this reason, it is necessary to store the field names (meta-data) in every document in the collection. When querying according to a certain field, only documents containing that field name are searched. If an index is not created on the fields included in the query, it is obvious that the query will run much slower on this schema independent structure than the relational database.

MongoDB automatically creates a unique key field named '_id', 12 bytes in size and hexadecimal type for each document added to the collection. The '_id' field is indexed automatically, just as the primary key is automatically indexed for each table created in relational databases.

2.1 Indexing in MongoDB

Like many relational database management systems, B-tree structure is used for indexing in MongoDB. Different indexing types such as Single Field, Compound, Multikey, Geospatial, Text and Hashed are supported. The difference of compound indexes from single field indexes is that they are created on more than one field. Multikey indexes, which are created on the fields that store an array, create a separate index entry for each element of the array, allowing documents containing the searched element or elements to be found. Hashed indexes are used for performing hash-based sharding. In accordance with the scope of our study, this section will focus on text indexes.

To create a text index on one or more fields containing text data, the "text" tag must be given in the 'createIndex' method. The following command creates a text index on the 'subject' and 'comments' fields in the 'reviews' collection:

```
db.reviews.createIndex({ subject: "text",
                         comments: "text" })
```

If the documents contain highly unstructured data, that is, if the field names are different in each of them; with the wildcard specifier ($**), all string type fields can be included in the text index:

```
db.reviews.createIndex({ "$**": "text" })
```

The text index stores an index entry for each unique stemmed word in the indexed fields of each document in the collection. Simple suffix stemming is used by discarding language-specific stop words (e.g. in English, the, and, a, etc.). In MongoDB 4.2, a text index can be created for one of 15 different languages. The default text indexing language is English. If you want to create an index according to a different language, the "default_language" option can be used:

```
db.books.createIndex({ content: "text" },
                     { default_language: "spanish" })
```

Since the index element is created according to the root of the word while making language-specific indexing, 'compute', 'computer', 'computation' and 'computing' searches are all searched with the 'comput' key and give the same result (The Snowball stemming algorithm used by MongoDB can be tried from: https://snowballstem.org/demo.html)

For example, suppose an English text index is created in the 'content' area of the 'books' collection. In the first query below, the number of documents containing the word 'computing' is requested using this index. Since 'compute' is indexed as a key, other words derived from this word will also be included in the result. For this reason, it will produce more results than the second query (regex type), which searches only those containing the word 'computing'.

```
db.books.find(
    { $text: { $search: "computing" } }).count()
db.books.find(
    { content: /computing/ }).count()
```

Only one text index can be created for a collection. As mentioned before, this index can contain many fields. In the first query that searches using a text index, if different fields were indexed besides 'content', the related words in those fields would also be included in the result. Therefore, there is no need to specify any fields when searching on text indexes.

Two main disadvantages can be mentioned for text indexes:

- They are generally large, since they contain every unique stemmed word in the indexed fields.
- Since the index needs to be updated after each document is added, the insertion times slow down as the number of documents increases.

2.2 Compression in MongoDB

The WiredTiger Storage Engine of MongoDB uses Snappy compression for all collections and prefix compression for all indexes by default. It is also possible to store collections and indexes without compression, but they are not preferred because it will have a significant negative impact on storage capacity and performance. Zlib and Zstd compression methods, which provide higher compression rates than Snappy, can also be used to compress the collections.

The Snappy compression library written in C++ by Google uses a variant of LZ77 algorithm. Since it aims fast compression and decompression rather than good compression ratio, it does not use any entropy encoder like Huffman or Arithmetic coding [1].

The Zlib method uses the DEFLATE algorithm [21] developed by Phil Katz for the PKZip compression tool. The DEFLATE algorithm, also used by 'gzip' file format, uses a combination of LZSS (a derivative of LZ77) and Huffman coding [22].

Zstd (Zstandard), which started to be used in MongoDB with version 4.2, was developed on Facebook by Yann Collet in 2015. It offers close compression ratio with Zlib while compressing and decompressing at speeds close to Snappy. It combines LZ77 with a large search window and a fast entropy coding stage, using both Finite State Entropy (FSE) and Huffman coding. FSE is a new kind of Entropy encoder that is based on Jarek Duda's ANS theory [23] and provides precise compression accuracy (such as Arithmetic coding) at much higher speeds. While there can be more than 20 times speed difference between the fastest compression mode and the slowest mode, decompression is fast in any case; ranges between the fastest and slowest modes by less than 20%.

3 MWCA

MWCA (Multi-Stream Word-Based Compression Algorithm) is a word-based text compression algorithm that stores the compressed data by dividing it into six different streams. The current version works with a semi-static model. In the first pass, all the different words in the text and their frequencies of occurrence are obtained and sorted from the most frequent to the least. The D1 dictionary is created from the most frequent 255 words and the D2 dictionary from the next 65536 words. Since a value (0) is used as an escape character in the first dictionary, it contains 256-1 words. Spaceless word model is used to obtain words [24]. A hash table is used to store the word list and provide O(1) access to this list during encoding. The size of this hash table was determined based on the Heaps' law [25].

In the second pass, compression is performed using dictionaries. If the word to be encoded is found in the D1 dictionary, the index number is stored in the S1 stream as one byte. If it is one of the words in the D2 dictionary, it is stored in the S2 stream with two bytes. If the word to be encoded doesn't exist in both dictionaries, it is stored in S3 stream in an uncompressed form. A bit vector (BV) is also created during coding. If the encoded word is found in the D1 dictionary and written to the S1 stream, a bit with a value of '0' is added to BV, and if it is found in the D2 dictionary and written to the S2 stream, a bit with a value of '1' is added to BV. With the help of the bit vector, the decoder can decide whether to read the next word from the S1 stream or the S2 stream. While encoding a word that is not found in the dictionaries, a '0' bit is written to BV and a '0' byte is written as an escape character to S1. If the decoder encounters '0' in S1, it will look at the S3 stream for the next word. At the end of the encoding, six different streams; D1, D2, S1, S2, S3 and BV are created as output. As an example, the streams generated during the compression of the title of this article with MWCA are shown in Figure 1.

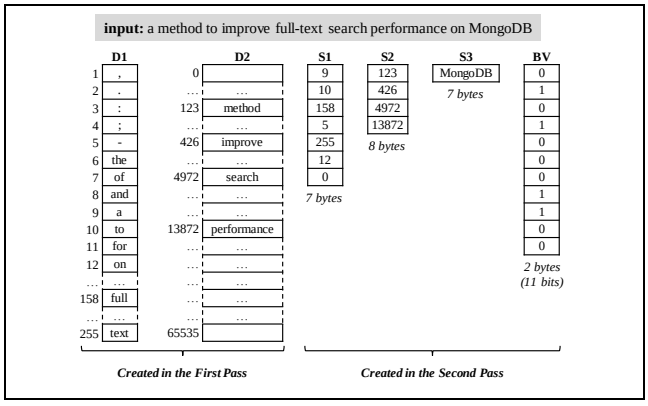

Figure 1. Streams generated when encoding with MWCA.

Since the given text example is small, if only that text was compressed, there will be only a few items in the D1 dictionary. To address all streams, it is assumed that a text large enough to fill the D1 and D2 dictionaries (containing 255 + 65536 different items) was compressed. In other words, while the D1 and D2 dictionaries in Figure 1 were created for a large text in the first pass, other streams created in the second pass show the compression of a line in that text. As can be seen in Figure 1, punctuation marks are included in the dictionary like words. The phrase "full-text" in our example is considered as three different words ('full', '-', 'text') and encoded as 158, 5 and 255. It is assumed that three of the other eight words ('a', 'to', 'on') are also in the D1 dictionary, four of them ('method', 'improve', 'search', 'performace') are in the D2 dictionary, and the word 'MongoDB' is not included in both dictionaries.

The MWCA decoder first checks the BV. In our example, since the first bit is '0', it looks at the stream S1 and reads the value 9. The word 'a' in the 9th position of the D1 dictionary is written to the output. Since the second bit in BV is '1', the code of the second word is taken from the S2 stream (123) and the word 'method' is found from the D2 dictionary and written to the output. The next words are found similarly and written to the output. Normally, a space character is inserted between words by the decoder. If a punctuation mark is to be written to the output, no space is automatically added before or after it. Therefore, "full-text" is not displayed as "full - text" with spaces in the output. Because the last bit in BV is '0', S1 is checked for the code of the last word. Since the last value in S1 is also '0', it is understood that this word does not exist in D1 and D2 dictionaries and should be obtained from S3 stream.

The smaller the size of the data to be compressed, the lower the compression rate of the compression algorithms because the similarities will be less in the data. Using a static dictionary in both the encoder and the decoder provides a better compression ratio for small files, rather than adding a dictionary next to each compressed file. It is important to use a static dictionary suitable for the content of the data to be compressed for a good compression ratio. For the Zstd compression algorithm to compress small files better, a static dictionary can be created by training according to a data set containing similar files (Facebook Zstd, The case for Small Data compression). Brotli compression algorithm developed by Google also uses a static dictionary [26].

When small text files are compressed with MWCA, the total size of D1 and D2 dictionaries will likely be more than the total size of S1 and S2 streams. As with other compression methods, MWCA's success in terms of compression ratio decreases with the size of the text to be compressed. The compression ratios of MWCA for texts of different sizes are given in Table 1. The ratios of the dictionaries, streams and bit vector created by MWCA according to the total size of the compressed file are given in Figure 2. As can be seen from the figure, S3 stream does not exist in text files smaller than 10 MB, since the number of different words is usually less than 65536 + 255. In even smaller text files, such as 1-2 KB, the D2 dictionary and S2 stream do not created as the number of different words will not exceed 255.

In this study, the benefits to be obtained by using only 3 of the 6 different streams created by the MWCA compression algorithm in the text index are emphasized. Our goal is not to propose an alternative to the compression methods used in MongoDB, but to recommend MWCA as a storage model that can be used with them. For this reason, the term *MWCA Model* is used in the next sections.

Table 1. Compression ratios of texts of different sizes with MWCA.

Text Size	Unique Words	Original Size (byte)	Compressed Size (byte)	Ratio (%)
1 KB	132	1,024	1,021	99.71
10 KB	762	10,240	8,222	80.29
100 KB	3,852	102,400	60,181	58.77
1 MB	14,316	1,048,576	447,637	42.69
10 MB	52,053	10,485,760	3,966,916	37.83
100 MB	199,290	104,857,600	37,659,581	35.91
1 GB	627,471	1,073,741,824	382,813,279	35.65

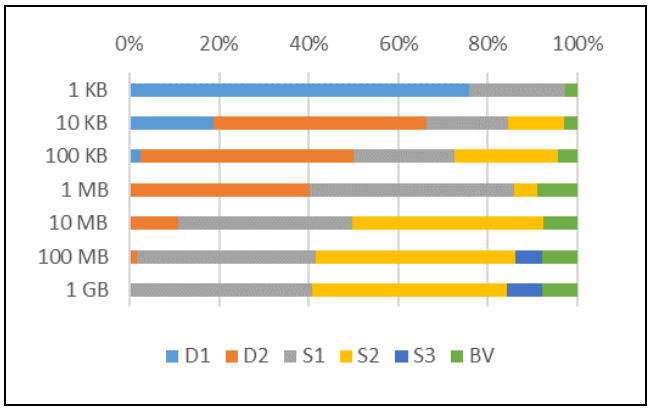

Figure 2. Ratios of dictionaries, streams, and bit vector relative to the total size of the compressed file.

4 Experimental results and advantages of the MWCA model

Each of the 21405 English books obtained from the website https://www.gutenberg.org is included as separate documents in the collection created for the experiments. When the collection is stored uncompressed, its total size is 10.002.726.912 bytes. MongoDB 4.2.2 Community database built on Windows 10 Enterprise 64-bit operating system was used in the experiments. The test computer specs: Intel (R) Xeon (R) CPU E3-1225 v3 @ 3.20GHz CPU, 16 GB RAM, 480 GB SSD drive.

4.1 Compression ratio advantage

Recompressing the data encoded by MWCA with another compression algorithm may give a better compression ratio than compression of the algorithm alone, especially when a large text is compressed. Table 2 shows the sizes and compression ratios when the test collection is compressed with MWCA, Snappy, Zlib and Zstd methods and recompressed with other methods after MWCA.

Table 2. Sizes of collections when different compression methods are used.

Compression	Size (byte)	Ratio (%)
-	10,002,726,912	100.00
MWCA	5,666,115,584	56.65
Snappy	6,108,823,552	61.07
MWCA + Snappy	4,642,598,912	46.41
Zlib	3,782,737,920	37.82
MWCA + Zlib	3,453,874,176	34.53
Zstd	3,553,533,952	35.53
MWCA + Zstd	3,572,006,912	35.71

As seen in Table 2, using the MWCA model together with Snappy gives 10% better results than MWCA's compression ratio and 15% better than Snappy's compression ratio. Although MWCA could not increase the compression ratio of Zstd, it increased the compression ratio of Zlib by 3% and enabling it to reach the best compression ratio. These 8 collections have been stored with different names for use in subsequent tests. The method of creating test collections is given in Figure 3.

4.2 Text index creation time and index size advantage

In 4 collections that do not use the MWCA model, each document has a 'name' field that stores the name of the file and a 'content' field that stores the content of the file. As MWCA divides the data into 6 streams, the other 4 collections using the MWCA model have D1, D2, S1, S2, S3 and BV fields instead of the 'content' field. D1, D2 and S3 fields to be used in the text index are stored as 'string' and the other 3 fields are stored as 'binary data'.

Text indexes are created by using the *createIndex({content: "text"})* method in 4 collections containing the 'content' field, and *createIndex({D1: "text", D2: "text", S3: "text"})* method in the other 4 collections. The size of these indexes, the number of keys they contain, and their creation times are given in Table 3. There are two phases in the creation of text indexes. The first of these is to scan the entire collection and find the key values, and the other is to add these keys to the index. The duration of the first phase is given as Collection Scan Time and the duration of the second phase as Key Insertion Time separately in the table. As can be seen in this table, since the total size of the *D1*, *D2* and *S3* fields is smaller than the 'content' field (*len(content) / len(D1+D2+S3) = 7.32*), the collection scan times are completed approximately 45% faster in each compression option. Although the number of keys in the MWCA model is approximately 3% less, both key insertion times and index sizes have decreased by 10%. Compared to the total indexing times, there was more than 40% time savings for each compression option.

The reason why there are fewer keys in the index of collections using MWCA model is that MongoDB's stemming algorithm does not apply stemming to words that have an underscore character at the end.

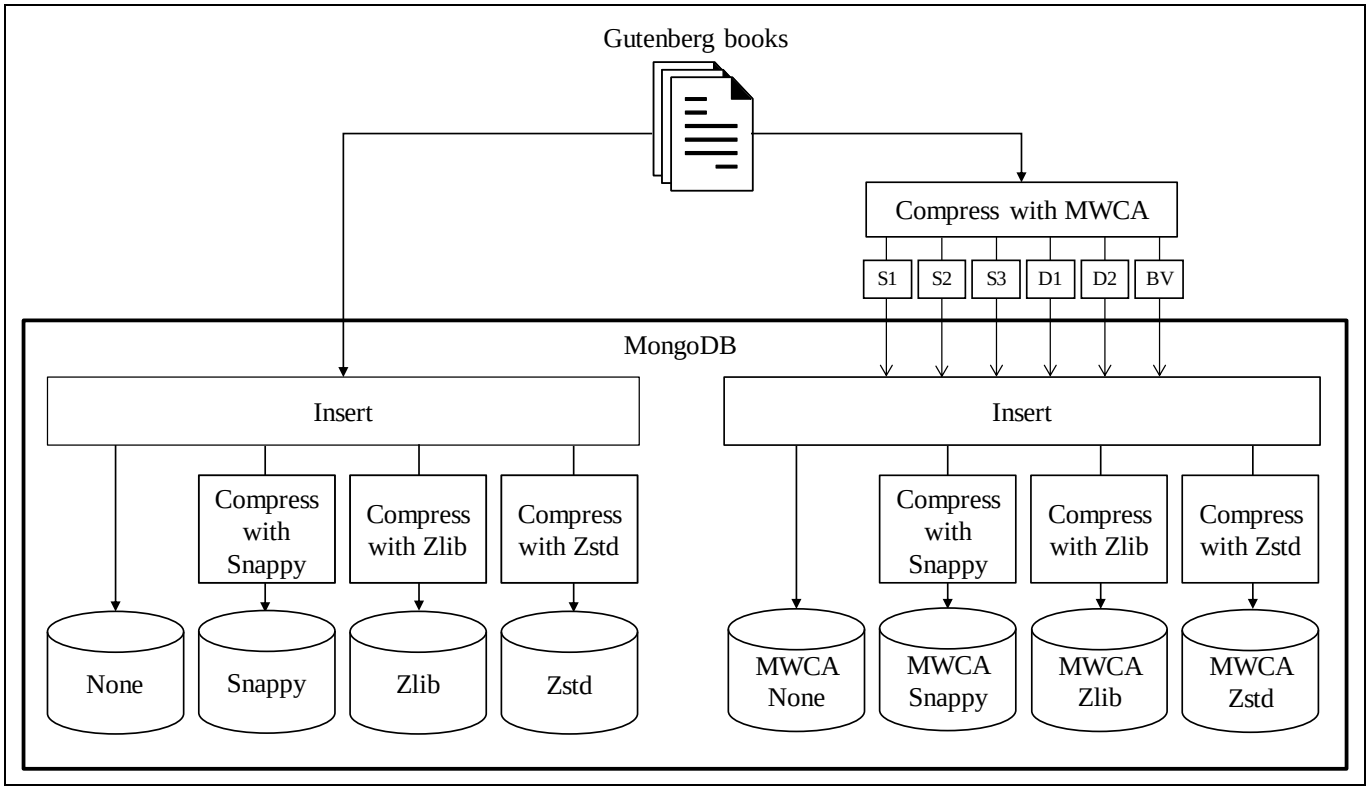

Figure 3. Streams generated when encoding with MWCA.

Table 3. Size, number of keys and creation times of text indexes.

Collection Name	Index Size (byte)	# of keys	Collection Scan Time (s)	Key Insertion Time (s)	Total Time (s)
None	1,563,652,096	109,623,048	845	104	949
MWCANone	1,398,439,936	106,606,886	471	95	566
Snappy	1,557,426,176	109,623,048	881	105	986
MWCASnappy	1,392,726,016	106,606,886	484	95	579
Zlib	1,557,426,176	109,623,048	931	104	1035
MWCAZlib	1,392,726,016	106,606,886	524	95	619
Zstd	1,557,426,176	109,623,048	924	105	1029
MWCAZstd	1,392,726,016	106,606,886	510	95	605

For example, EnglishStemmer produces 'mobil' output for the word 'mobile', while it produces 'mobile_' output for 'mobile_' (it can be checked from: Demo - Snowball). The sections written in italic in certain parts of the books on the Gutenberg.org are given between the two '_' symbols while being stored in plain text. For example: The phrase written as '*primum mobile*' in the HTML version of the book "Beacon Lights of History, Volume I" is stored as '_primum mobile_' in the TXT version. If there are adjacent punctuation marks at the beginning or end of the word, MWCA separates them from the word and adds each one as a different word to the dictionary. For this reason, while 'mobil' and 'mobile_' are stored as different keys in the text index created from the 'content' field, there is a single key (mobil) in the text index created from the "*D1*, *D2* and *S3*" fields. In the book mentioned, the word 'mobile' only exists in one place and has '_' at the end. Therefore, in collections that do not use the MWCA model, this book can only be found by searching 'mobile_', it cannot be found when searching 'mobile'. On the collections using MWCA model, it can be said that more accurate search results are obtained since the relevant book can be found when this search is made without writing an underscore.

As can be seen in Table 1, normally text files smaller than 10 MB will not have data in the *S3* stream. Although all the 21405 books in our test are below this size, only the following 2 books have data in the *S3* stream because they contain many different numbers:

- The First 100,000 Prime Numbers (https://www.gutenberg.org/files/65/65.txt),
- Pi to 1,000,000 places (https://www.gutenberg.org/files/50/50.txt).

If the documents in a collection do not contain such unusual text, *S3* stream may not be required. Since the document size that can be added to a collection in MongoDB is limited to 16 MB, in most cases it would not be a problem not to use *S3* in the MWCA algorithm or not to include it in the text index.

4.3 Search time and memory usage advantage for text index

In Table 4, the 100 words selected for the search test are listed according to how many different documents they exist in. As mentioned in the previous section, collections that do not use the MWCA model cannot find some words due to underscore problems. The '+' column shows how many more documents are found in the collections using the MWCA model, compared to other collections. As an example, the word 'mobile' is found in 3684 documents in collections that do not use the MWCA model, while it is found in 3684 + 113 = 3797 documents in collections using this model. While obtaining these values, the following statement was used:

```
db.[collectionName].find(
                {$text:{$search: [word]}}).count()
```

When a query is sent to the MongoDB server, if the required index and data (working set) are not available in RAM, they are first copied from disk to memory. There is no setting in MongoDB to keep an index permanently in memory. When a word that exists in all documents in a collection is queried, a working set covering almost the entire collection is written into memory. Queries sent later return much faster results as they will use the data available in RAM. To ensure that the search for the first words in the test was not slow due to disk access, the word 'set', one of the words found in all documents, was searched before the test to transfer most of the required data to RAM. When the server cannot find enough memory space to process the query, it needs to clean the old data with the LRU method, which slows down the response time. For a fair comparison, the service was restarted before testing on each collection. The test was carried out with a Python code that first queries the word 'set', then the other 100 words in the order given in Table 4.

Table 4. Words used in the search test.

Word	Count	+	Word	Count	+	Word	Count	+	Word	Count	+
allows	21405	0	conclusion	16509	12	coined	10486	23	psychology	4563	24
computer	21388	0	retiring	16305	38	bowling	10391	28	Turkish	4297	19
original	21331	0	judgment	16249	5	reviews	10387	384	mobile	3684	113
turns	21187	2	description	16153	10	compete	10182	2	actress	3298	3
staff	21029	1	solely	15826	35	averaged	10167	14	Denmark	3047	25
feels	20837	5	corresponds	15505	1	knights	9796	40	Norway	2877	29
fired	20518	7	appreciation	15392	4	muscles	9790	7	Sweden	2832	30
attachment	20319	15	enables	15276	0	electrical	9631	16	anthem	2094	9
growing	20260	3	profession	15204	24	China	9533	35	broadcast	1909	2
watch	20161	17	amazing	14775	57	Italy	9345	85	Jerome	1885	19
earth	19958	4	capitalism	14742	23	Germany	8839	43	Hannah	1591	28
studied	19300	10	assembly	14510	30	Spain	8340	63	phone	1374	15
anyone	19038	0	abroad	14285	11	eventually	8118	4	rehabilitation	1164	1
quarter	18829	52	host	14146	23	champion	7885	19	spherical	1071	4
apart	18744	11	France	13744	61	Mexico	7413	26	Balkan	855	1
crowd	18450	7	characteristics	13727	8	Egypt	7338	51	Finland	842	7
ranges	18148	27	religious	13629	30	Turkey	7015	23	Bulgaria	706	2
supplies	18021	9	artist	13136	25	grandmother	6806	14	traction	557	2
accompanying	17919	8	ocean	13109	44	Holland	6404	36	ethnic	484	3
pull	17478	47	farming	12875	20	Russia	6303	24	vocalist	413	0
proposes	17245	7	achievement	12561	45	Greece	5788	54	coastal	348	0
seasons	17214	24	overlooking	12250	19	adjacent	5525	3	television	209	0
handles	17072	18	student	11989	7	voluntarily	5440	18	conceptual	156	1
grand	16984	57	divisions	11848	3	massacre	5327	17	database	64	0
England	16630	48	farmer	11242	26	devastated	4998	7	Edirne	18	0

The results of the search test are given in Table 5. The search time of the word 'set' is given in the first line. It took between 32 and 64 seconds to get the result, as the working set was loaded into RAM with this search. Although the word 'allows' is also included in all documents, the search time has been much faster since most of the working set is in RAM. To simplify the results, 4 of the 100 words in the test are included from the top, 4 from the middle and 4 from the end. The 'Total' line below is the sum of the search times of these 100 words (not including the time for 'set'). The last line shows how many times the MWCA model is faster than the currently used structure. As seen in Table 5, the frequency of the searched word (count) affects the query time. It is also seen that the MWCA model provides a higher rate of time difference in words with high frequency.

Figure 4 shows how much space the mongod service uses from RAM during the search for both the word 'set' and the other 100 words. It is seen that using the MWCA model provides 40%, 37%, 34% and 34% less memory usage for None, Snappy, Zlib and Zstd respectively.

When the *mongod* is started with default settings, WiredTiger Storage Engine uses "50% of (RAM - 1 GB)" memory (MongoDB Documentation: WiredTiger Storage Engine, Memory Use). Since our test machine has 16 GB of RAM, it can use 7.5 GB of memory when started with default settings. As seen in Figure 4, collections that do not use the MWCA model need more than 8 GB of RAM. For this reason, during the search process, the server frequently deletes the old data from RAM with the LRU method and loads the new data from the disk. In our initial tests, we started the *mongod* service with default settings and observed that the results produced by collections not using the MWCA model were 2 times slower than the corresponding results in Table 5. In order not to give any advantage to the collections using the MWCA model in this respect, while obtaining the results given in Table 5, the *mongod* service was started using the expression "--wiredTigerCacheSizeGB=10" so that the service can use 10 GB of RAM. It is seen in Table 5 that the combination of Snappy block compression, which is the default storage method of WiredTiger Storage Engine, and the MWCA model can be searched 51 times faster. This rate is more than 100 times with other compression options. It is also an important advantage that using MWCA model provides 34% to 40% less RAM usage. Although the Snappy algorithm has a lower compression ratio compared to Zlib and Zstd, it is seen that it is much faster when searching in the text index. The result obtained with Snappy was still 3 times slower than storage without compression. The results obtained by the collections of MWCA model are around 3.5 seconds in each compression option. Therefore, instead of storing it with no compression or low compression with Snappy, when using higher compression with Zlib or Zstd using MWCA model, there will be no speed disadvantage in searching the text index.

The main reason for the MWCA model to achieve a high rate of speed gain is that S1, S2 and BV fields, which store most of the data in the documents, are stored as 'binary data'. If these fields are stored as 'string' like other fields, the search speed will be much slower as they will be included in the working set when searching on the text index.

4.4 Search time and memory usage advantage for regex queries

Searching for a specific word in a collection can also be done with regex query without using text index. The speed of this type of search is slower because all data must be scanned. As the total size of the *D1*, *D2* and *S3* fields is less than the size of the 'content' field, the MWCA model will also be useful in searching with regex.

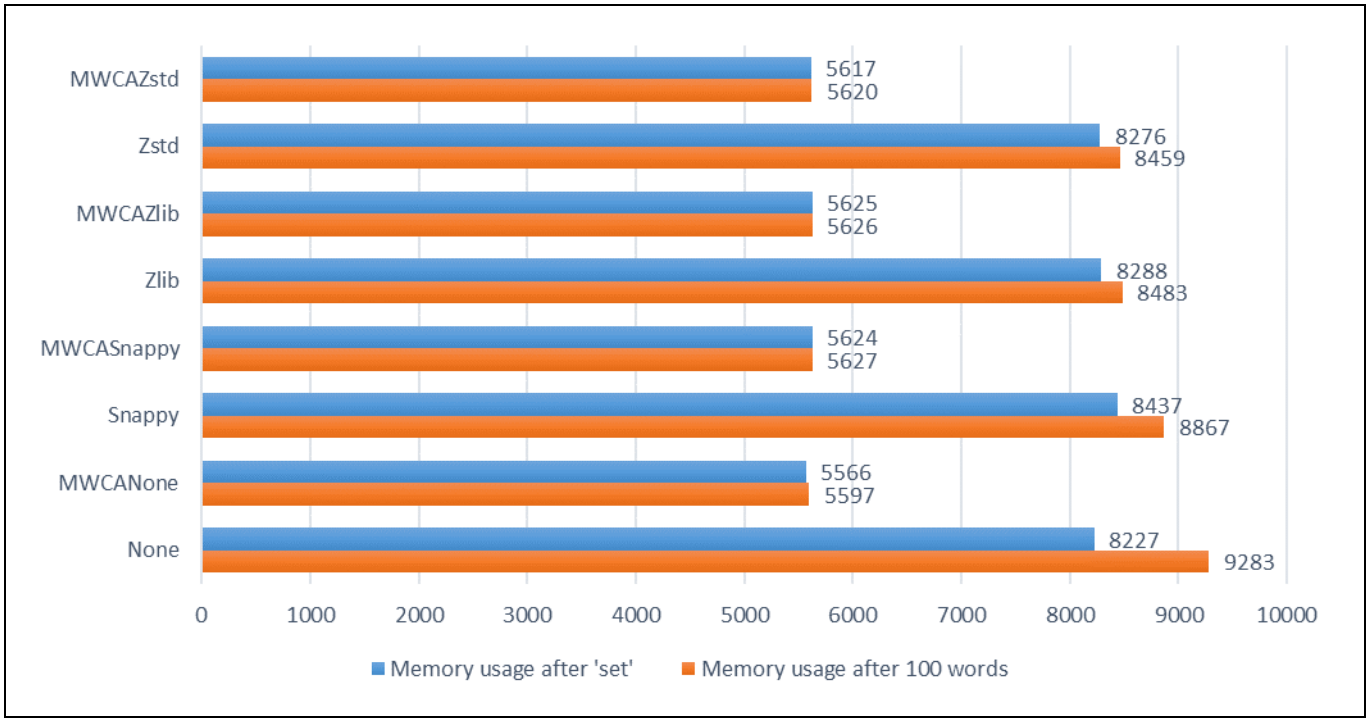

Figure 4. Memory usage results of the test using text index (MB).

The first of the queries below was carried out in collections that did not use the MWCA model, and the second in collections using it, and the results in Table 6 were obtained.

```
db.[collectionName].find(
{ content: {$regex: [word] }}).count()
```

```
db.[collectionName].find(
{ $or: [{D1: {$regex: [word] }},
        {D2: {$regex: [word] }},
        {S3: {$regex: [word] }}]}).count()
```

Memory usage in regex queries is given in Figure 5. When Figure 4 and 5 are examined together, it is seen that the memory usage of the collections using MWCA model is similar in both tables. Collections that do not use the MWCA model used slightly less memory in the regex query. The contribution of the MWCA model in terms of memory usage was between 29% and 34% in regex queries.

This test was done with 10 of the 100 words used in the previous test. Unlike Table 5, Table 6 also includes the 'Count' column, where the result values of the query are shown. Since there is no stemming operation in regex queries, the values in this column are different from the values of the same words in Table 4, and collections using the MWCA model produce the same count values. Because the entire collection is scanned, how many different documents contain the searched word does not affect the search time. The word 'television', which exist in only 201 documents, was the slowest found in all collections. The reason why the word 'quarter' is found the fastest and the word 'television' the slowest is probably related to the low frequency of the letter 'q' and the high frequency of the letter 't' in English. As mentioned in Section 4.2, the total size of the D1, D2 and S3 fields of all documents in the collection using MWCA model is 7.32 times less than the total size of the 'content' fields in the collection that does not use this model. The 7.8 times search speed difference between *None* and *MWCANone* is not surprising as it is close to the difference in the size of the scanned data. In other collections using block compression, the speed difference is even higher since those using the MWCA model decompress smaller data than those who do not.

According to the time results in both Table 5 and Table 6, the success order of collections that do not use the MWCA model is None, Snappy, Zstd and Zlib from best to worst. The collections using the MWCA model in both search tests gave close time results to each other (approximately 3.5 seconds in Table 5 and 25 seconds in Table 6). Whether compression was applied to the collection or which compression method was used did not affect the results.

5 Conclusion

In our previous study, it was shown that words can be searched quickly without decompression on texts compressed with the MWCA compression algorithm [4]. Searching in D1, D2 and S3 streams instead of the entire text as we mentioned in the section on regex queries is similar to the approach presented in that study. In this study, we have demonstrated the advantages of using its multi-stream structure with indexes through our tests on MongoDB. While creating a text index in MongoDB, instead of using the entire large text such as book content for the index, encoding that text with MWCA and indexing three of the six streams provides advantages from various aspects. According to the results of our tests:

- Index creation time is reduced by approximately 40%.
- The size of the index is reduced by approximately 10%.
- In a query such as "How many documents contain a given word", it gives up to 150 times faster results and uses 34% to 40% less memory.

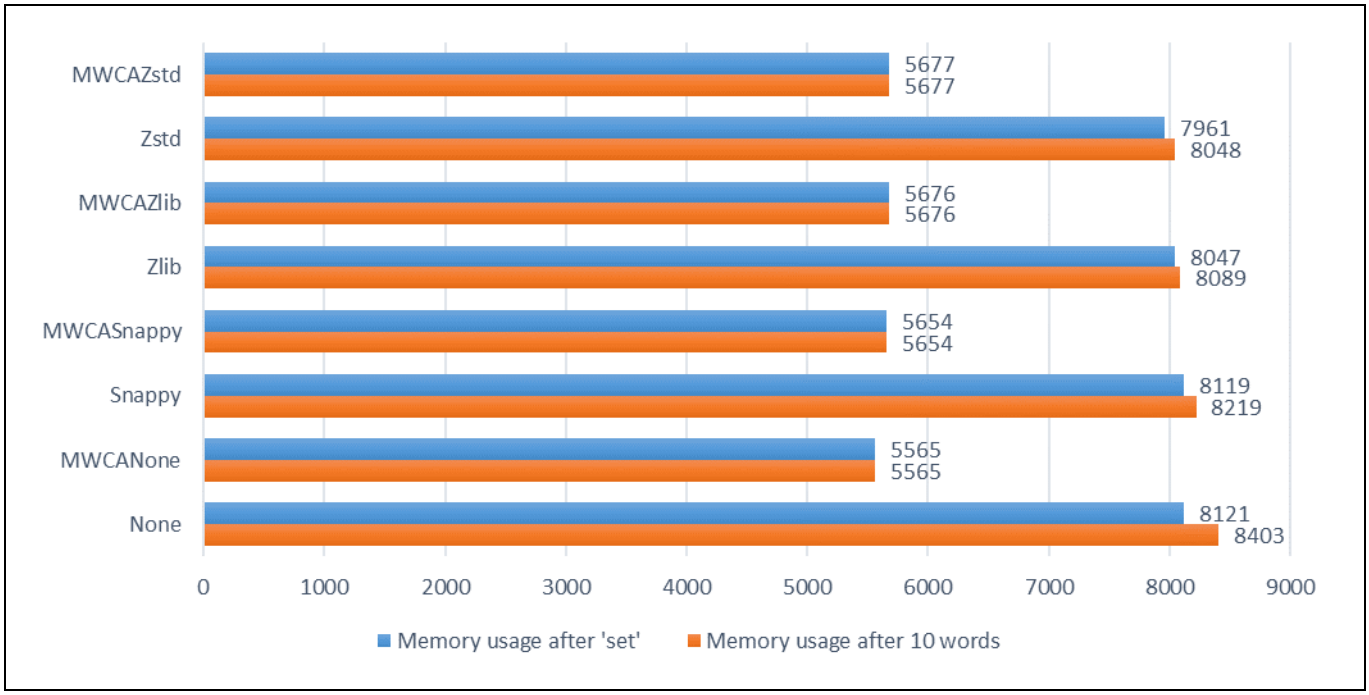

Figure 5. Memory usage results of the test using regex (MB).

Table 5. Time results of the search test on the text index (ms).

Word	None	MWCA None	Snappy	MWCA Snappy	Zlib	MWCA Zlib	Zstd	MWCA Zstd
set	44286	32463	46396	32955	64488	54995	55032	41140
allows	7731.03	89.98	5623.16	86.02	11318.58	84.95	6982.75	116.09
computer	1698.65	61.23	3626.31	63	9189.04	59.1	6950	60.95
original	3004.18	63.79	3405.95	61.88	9252.1	61.96	8200.31	60.97
turns	1397.03	64.96	3167.36	63.31	8697.17	59.46	6653.19	59.94
…	…	…	…	…	…	…	…	…
China	548.17	37.98	1495.93	33.98	4567.38	35.98	3473.37	33.98
Italy	449.21	33.99	1683.12	31	4608.85	34.97	3605.28	30.98
Germany	299.85	31.41	1841.4	31.99	4593.86	31.98	3080.24	31.98
Spain	460.24	28.2	1586.14	28.66	4246.09	28.98	2951.99	28.98
…	…	…	…	…	…	…	…	…
television	4	3	25.98	4	53.96	3.01	13	3.53
conceptual	7.01	1.84	16.98	3.03	40.98	3.01	35.22	4.01
database	4.99	2	5.01	2.77	8.99	2.01	7	3.01
Edirne	2	1.72	1.01	2.01	2.01	2	3	1.59
Total	66587.44	3451.52	182287	3572.31	500731.63	3414.47	385733.56	3480.6
Speed increase rate	19.29 times		51.03 times		146.65 times		110.82 times	

Table 6. Time results of the search test with regex (ms).

Word	Count	None	MWCA None	Snappy	MWCA Snappy	Zlib	MWCA Zlib	Zstd	MWCA Zstd
set	21405	28112	14669	29258	12216	61755	32674	49199	18101
computer	21385	20619.51	1868.51	19966.52	1921.92	26456.81	1902.47	27787.06	1902.85
quarter	18899	7938.91	969.45	14664.52	1014.49	16297.31	988.43	19960.64	982.48
description	15706	12413.66	1895.93	18326.16	1997.86	29044.45	1917.92	24133.1	1905.92
ocean	12261	18644.05	2487.58	24644.54	2630.99	36911.79	2541.55	31076.93	2504.57
farmer	10720	12260.6	1550.1	22917.07	1587.11	26923.65	1578.1	25203.07	1569.18
champion	7771	14003.71	2409.62	21919.53	2519.55	29437.34	2520.56	26247.05	2464.59
mobile	4842	15212.35	2124.31	19654.52	2198.76	33789.27	2151.39	28581.57	2123.8
anthem	2674	31750.18	4127.14	36538.55	4285.06	48292.87	4104.68	42764.67	4067.68
television	201	37030.03	4185.15	39957.46	4496.43	51652.78	4259.56	48300.5	4206.6
database	63	23244.1	3154.2	26562.6	3215.17	37852.44	3106.24	35483.43	3111.21
Total		193117.1	24771.99	245151.5	25867.34	336658.7	25070.9	309538	24838.88
Speed increase rate		7.80 times		9.48 times		13.43 times		12.46 times	

All of the tests in our study were carried out on the server machine. When queries are sent over a local client, the query times will not increase too much, since the transmission will be fast. If you connect to the server from a remote client, there will be much more additional network transmission time for each word queried. Assuming an average transmission time of 30 ms per word, the total times given at the bottom of Table 5 will increase by 3 seconds and the "Speed increase rate" values will be less. When 'name' fields are requested from a remote client instead of a count query, the transmission time varies depending on the word queried (only 18 names will be sent for the word 'Edirne', while 16630 names will be sent for 'England'). If the 'content' fields (or 6 fields used instead of 'content' in the MWCA model) are requested, the transmission time will be much longer. However, as an example, requesting the entire content of books containing the word 'England' is not a meaningful query as it requires a data transfer of GBs size. If an application with a MWCA decoder is used on the client side to query the books and download their contents, the data in compressed form can be sent 2 times faster and it may be possible to save time if the decoding time is not very long (depending on the power of the client machine).

Apart from the advantages it provides for the text index, the MWCA model also has advantages in the following situations:

- It gives approximately 10 times faster results in queries made with regex method without using index and uses 29% - 34% less memory,
- Increases the compression ratio of Snappy by 15% and compression ratio of Zlib by 3%.

The most important factor in the success of the proposed method is that it stores most of the data in binary type and MongoDB does not include this part in the working set. This structure not only increases search speed but also reduces memory usage. If a large number of new documents (books) are added, 10GB *wiredTigerCacheSize* will not be enough for collections that do not use the MWCA model and their query performance will decrease significantly. On the other hand, collections using the MWCA model will be able to maintain their performance even for much more documents as they use less memory. It can also be said that the performance difference of the proposed model would be much greater if we had not used a large enough *wiredTigerCacheSize* for all collections in our tests by starting the server with the default settings.

6 Author contribution statements

Altan MESUT contributions: Formation of the idea, literature review, performing the tests, evaluation of the results and writing the article.

Emir ÖZTÜRK contributions: Literature review, obtaining data for tests, creation of test environment and program codes.

7 Ethics committee approval and conflict of interest statement

There is no need to obtain permission from the ethics committee for the article prepared and there is no conflict of interest with any person / institution in the article prepared.

8 References

[1] Qiao Y. An FPGA-Based Snappy Decompressor-Filter. MSc Thesis, Delft University of Technology, Delft, Netherlands, 2018.

[2] Deutsch P, Gailly JL. "Zlib compressed data format specification version 3.3". RFC 1950, USA, 1996.

[3] Collet Y, Kucherawy M. "Zstandard compression and the application/zstd Media Type". RFC 8478, USA, 2018.

[4] Öztürk E, Mesut A, Diri B. "Multi-Stream word-based compression algorithm for compressed text search". *Arabian Journal of Science and Engineering*, 43(12), 8209–8221, 2018.

[5] Habib A, Islam MJ, Rahman MS. "A dictionary-based text compression technique using quaternary code". *Iran Journal of Computer Science*, 3(3), 127–136, 2020.

[6] Rahman MA, Hamada M. "Burrows–Wheeler transform based lossless text compression using keys and huffman coding". *Symmetry*, 12(10), 1654-1667, 2020.

[7] Mahmood MA, Hasan KMA. "Efficient compression scheme for large natural text using zipf distribution". *International Conference on Advances in Science, Engineering and Robotics Technology*, Dhaka, Bangladesh, 3 May 2019.

[8] Bharathi K, Kumar H, Fairouz A, Al Kawam A, Khatri SP. "A plain-text incremental compression (pic) technique with fast lookup ability". *IEEE 36th International Conference on Computer Design*, Orlando, FL, USA, 7-10 October 2018.

[9] Buluş HN, Carus A, Mesut A. "A new word-based compression model allowing compressed pattern matching". *Turkish Journal of Electrical Engineering & Computer Sciences*, 25(5), 3607–3622, 2017.

[10] Morishima S, Matsutani H. "Performance evaluations of document-oriented databases using GPU and cache structure". *IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, 20-22 August 2015.

[11] Kelec A, Dujlovic I, Obradovic N. "One approach for full-text search of files in MongoDB based systems". *IEEE 18th International Symposium INFOTEH-JAHORINA*, East Sarajevo, Bosnia & Herzegovina, 20-22 March 2019.

[12] Truica CO, Boicea A, Radulescu F. "Building an inverted index at the dbms layer for fast full text search". *Journal of Control Engineering and Applied Informatics*, 19(1), 94-101, 2017.

[13] Zobel J, Moffat A. "Inverted files for text search engines". *ACM computing surveys (CSUR)*, 38(2), 1-56, 2006.

[14] Greca S, Kosta A, Maxhelaku S. "optimizing data retrieval by using mongodb with elasticsearch". *International Conference on Recent Trends and Applications in Computer Science and Information Technology*, Tirana, Albania, 23-24 November 2018.

[15] Han L, Zhu L. "Design and implementation of elasticsearch for media data". *IEEE International Conference on Computer Engineering and Application*, Guangzhou, China, 18-20 March 2020.

[16] Lu W, Zhu L, Duan S. "Research and implementation of big data system of social media". *IEEE/ACIS 17th International Conference on Computer and Information Science*, Singapore, 6-8 June 2018.

[17] Mongodb, Inc. "Getting started with MongoDB Atlas Full-Text Search". https://www.mongodb.com/blog/post/getting-started-with-mongodb-atlas-fulltext-search (10.07.2021).

[18] Eyada MM, Saber W, El Genidy MM, Amer F. "Performance evaluation of iot data management using mongodb versus MySQL databases in different cloud environments". *IEEE Access*, 8, 110656-110668, 2020.

[19] Mehmood E, Anees T. "Performance analysis of not only SQL semi-stream join using MongoDB for real-time data warehousing". *IEEE Access*, 7, 134215-134225, 2019.

[20] Nurseitov N, Paulson M, Reynolds R, Izurieta C. "Comparison of JSON and XML data interchange formats: a case study". *22nd International Conference on Computer Applications in Industry and Engineering*; San Francisco, CA, USA, 4-6 November 2009.

[21] Deutsch P. "DEFLATE Compressed Data Format Specification Version 1.3". RFC 1951, USA, 1996.

[22] Deutsch P. "GZIP file Format Specification Version 4.3". RFC 1952, USA, 1996.

[23] Duda J, Tahboub K, Gadgil NJ, Delp EJ. "The use of asymmetric numeral systems as an accurate replacement for Huffman coding". *IEEE Picture Coding Symposium*, Cairns, QLD, Australia, 31 May-3 June 2015.

[24] Moffat A. "Word-based text compression". *Software: Practice and Experience*, 19(2), 185-198, 1989.

[25] Heaps HS. *Information Retrieval: Computational and Theoretical Aspects*. 1st ed. Sea Harbor Drive Orlando, FL, USA, Academic Press, 1978.

[26] Alakuijala J, Farruggia A, Ferragina P, Kliuchnikov E, Obryk R, Szabadka Z, Vandevenne L. "Brotli: A general-purpose data compressor". *ACM Transactions on Information Systems*, 37(1), 1-30, 2018.