Proximity Full-Text Search with a Response Time Guarantee by Means of Additional Indexes with Multi-Component Keys

© Alexander B. Veretennikov
Ural Federal University,
Yekaterinburg, Russia
alexander@veretennikov.ru

Abstract. Full-text search engines are important tools for information retrieval. In a proximity full-text search, a document is relevant if it contains query terms near each other, especially if the query terms are frequently occurring words. For each word in the text, we use additional indexes to store information about nearby words at distances from the given word of less than or equal to *MaxDistance*, which is a parameter. We had shown that additional indexes with three-component keys can be used to improve the average query execution time up to 94.7 times if the queries consist of high-frequency used words. In this paper, we present a new search algorithm with even more performance gains. We also present results of search experiments, which show that three-component key indexes enable much faster searches in comparison with two-component key indexes.

Keywords: full-text search, search engines, inverted indexes, additional indexes, proximity search, term proximity.

1 Introduction

A search query consists of several words. The search result is a list of documents containing these words.

In [11], we discussed a methodology for high-performance proximity full-text searches and a search algorithm. With the application of additional indexes [11], we improved the average query processing time by a factor of 94.7 for the case when the queries consist of high-frequency used words.

In this paper, we present the following new results.

1. We present a new search algorithm, with which we can improve the performance even more than it is improved in [11].
2. We present the results of search experiments that prove that three-component key indexes can be used to improve the average query execution time up to 12.93 times in comparison with two-component key indexes for the case when the queries consist of high-frequency used words.

In modern full-text approaches, it is important for a document to contain search query words near each other in order to be relevant to the context of the query, especially if the query contains frequently used words. The impact of the term-proximity is integrated into modern information retrieval models [2, 8, 9, 19].

Words appear in texts at different frequencies. The typical word frequency distribution is described by Zipf's law [20]. An example of words' occurrence distribution is shown in Figure 1. The horizontal axis represents different words in decreasing order of their occurrence in texts. On the vertical axis, we plot the number of occurrences of each word.

Proceedings of the XX International Conference "Data Analytics and Management in Data Intensive Domains" (DAMDID/RCDL'2018), Moscow, Russia, October 9-12, 2018

The full-text search task can be solved with inverted indexes [6, 10, 21]. With ordinary inverted indexes, for each word in the indexed document, we store in the index the record (*ID*, *P*), where *ID* is the identifier of the document and *P* is the position of the word in the document. Let *P* be an ordinal number of the word in the document.

For proximity full-text searches, we need to store the (*ID*, *P*) record for all occurrences of any word in the indexed document. These (*ID*, *P*) records are called "postings".

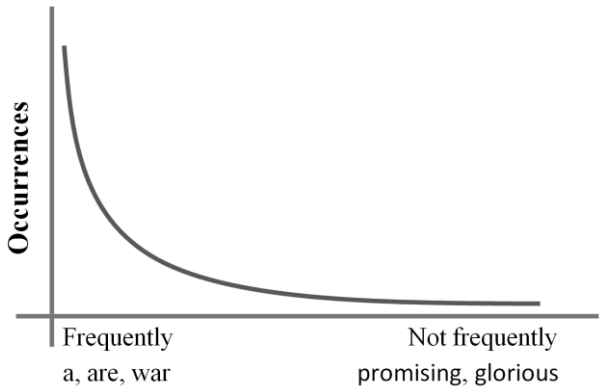

Figure 1 Example of a word frequency distribution.

Therefore, the query search time is proportional to the number of occurrences of the queried words in the indexed documents.

Consequently, to evaluate a search query that contains high-frequency occurring words, a search system needs much more time (see Figure 1, on the left side) than a query that contains ordinary words (see Figure 1, on the right side).

A full-text query is a "simple inquiry", and accordingly [7], to prevent the interruption of the thought

continuity of the user, the query results must be produced within two seconds. In this context, we present the following problem. It is common to have a full-text search engine that can usually evaluate a query within 1 sec. of time. However, it works very slowly, for example, requiring 10-30 sec., for a query that contains frequently occurring words.

We can illustrate this problem by the following example. We downloaded pgdvd042010.iso from the Project Gutenberg web page, which contains their files as of April 2010, and indexed its content using Apache Lucene 7.4.0 and Apache Tika 1.18. We indexed approximately 64 thousand documents with a total length of approximately 13 milliard characters (a relatively small number). We indexed all words. Then, we evaluated the following queries using the equipment from section 4.1 of the current paper:

"Prince Hamlet"~4 – this search took 172 milliseconds, and

"to be or not to be"~4 – this search took 21 seconds.

The suffix "~4" instructs Lucene to search such texts in which the queried words contain no more than 4 other words between them.

To improve the search performance, early-termination approaches can be applied [1, 4]. However, early-termination methods are not effective in the case of proximity full-text searches [11]. It is difficult to combine the early-termination approach with the integration of term-proximity information into relevance models.

Another approach is to create additional indexes. In [3, 18], the authors introduced some additional indexes to improve the search performance, but they only improved phrase searches.

With our additional indexes, an arbitrary query can be evaluated very fast.

2 Lemmatization and lemma type

2.1 Word type

In [12], we defined three types of words.

Stop words: Examples include "and", "at", "or", "not", "yes", "who", "to", and "be". In a stop-words approach, these words are excluded from consideration, but we do not do so. In our approach, we include information about all words in the indexes.

We cannot exclude a word from the search because a high-frequency occurring word can have a specific meaning in the context of a specific query [11, 18]; therefore, excluding some words from consideration can induce search quality degradation or unpredictable effects [18].

Let us consider the query example "who are you who". The Who are an English rock band, and "Who are You" is one of their songs. Therefore, the word "Who" has a specific meaning in the context of this query.

Frequently used words: These words are frequently encountered but convey meaning. These words always need to be included in the index.

Ordinary words: This category contains all other words.

2.1 Lemmatization

We employ a morphological analyzer for lemmatization. For each word in the dictionary, the analyzer provides a list of numbers of lemmas (i.e., basic or canonical forms). For a word that does not exist in the dictionary, its lemma is the same as the word itself.

Some words have several lemmas. For example, the word "mine" has two lemmas, namely, "mine" and "my".

We define three types of lemmas: stop lemmas, frequently used lemmas and ordinary lemmas. We sort all lemmas in decreasing order of their occurrence frequency in the texts. We call this sorted list the *FL*-list. The number of a lemma in the *FL*-list is called its *FL*-number. Let the *FL*-number of a lemma *w* be denoted by *FL*(*w*).

The first *SWCount* most frequently occurring lemmas are stop lemmas. The second *FUCount* most frequently occurring lemmas are frequently used lemmas. All other lemmas are ordinary lemmas.

SWCount and *FUCount* are the parameters. We use *SWCount* = 700 and *FUCount* = 2100 in the experiments presented.

If an ordinary lemma *q* occurs in the text so rarely that *FL*(*q*) is irrelevant, then we can say that *FL*(*q*) = ~. We denote by "~" some large number.

2.2 Index type

We create indexes of different types for different types of lemmas. Let *MaxDistance* be a parameter that can take a value of 5, 7 or even greater.

The expanded (*f*, *s*, *t*) index or three-component key index [11, 16] is the list of occurrences of the lemma *f* for which lemmas *s* and *t* both occur in the text at distances less than or equal to *MaxDistance* from *f*.

We create an expanded (*f*, *s*, *t*) index only for the case in which $f \le s \le t$. Here, *f*, *s*, and *t* are all stop lemmas.

Each posting includes the distance between f and s in the text and the distance between f and t in the text.

The expanded (*w*, *v*) index or two-component key index [13, 14, 15] is the list of occurrences of the lemma *w* for which lemma *v* occurs in the text at a distance less than or equal to *MaxDistance* from *w*.

The lemma types considered are as follows: for *w*, frequently used, and for *v*, frequently used or ordinary. Each posting includes the distance between *w* and *v* in the text.

Other types of additional indexes are described in [11].

3 A new search algorithm

3.1 The search algorithm general structure

Our search algorithm is described in Figure 2.

Let us consider the search query "who are you who". After lemmatization, we have the following query:
[who] [are, be] [you] [who]. The word "are" has two lemmas in our dictionary.

With *FL*-numbers:

[who: 293] [are: 268, be: 21] [you: 47] [who: 293].
To use three-component key indexes, this query must be divided into two subqueries [11]:
Q1: [who: 293] [are: 268], [you: 47] [who: 293], and
Q2: [who: 293] [be: 21], [you: 47] [who: 293].

We can say that lemma "who" > "you" because *FL*(who) = 293, *FL*(you) = 47, and 293 > 47. We use the *FL*-numbers to establish the order of the lemmas in the set of all lemmas.

In [11], we defined several query types depending on the types of lemmas they contain and different search algorithms for these query types. The query does not need to be divided into a set of subqueries for all query types.

In this paper, we consider subqueries that consist only of stop lemmas.

After step 2, we evaluate the subqueries in the loop.

After all subqueries are evaluated, their results need to be combined into the final result set.

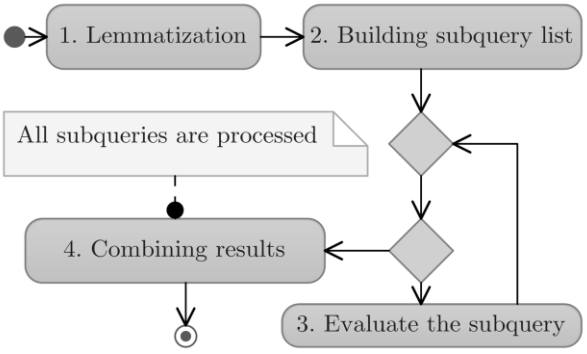

Figure 2 UML diagram of the search algorithm general structure.

3.2 Subquery evaluation

The algorithm of the subquery evaluation, when the subquery consists of only stop lemmas, is described in Figure 3.

We need to select the three-component key indexes required to evaluate the subquery.

For all selected indexes, we need to create an iterator object. The iterator object for the key (*f*, *s*, *t*) is used to read the posting list of the (*f*, *s*, *t*) key from the start to the end. The iterator object *IT* has the method *IT.Next*, which reads the next record from the posting list.

The iterator object *IT* has the property *IT.Value* that contains the current record (*ID*, *P*, *D1*, *D2*). Consequently, *IT.Value.ID* is the *ID* of the document containing the key, and *IT.Value.P* is the position of the key in the document.

For two postings *A* = (*A.ID*, *A.P*, *A.D1*, *A.D2*) and *B* = (*B.ID*, *B.P*, *B.D1*, *B.D2*), we define that *A* < *B* when one of the following conditions are met: *A.ID* < *B.ID* or (*A.ID* = *B.ID* and *A.P* < *B.P*).

The records (*ID*, *P*, *D1*, *D2*) are stored in the posting list for the given key in increasing order.

The goal of the *Equalize* procedure is to ensure that all iterators have an equal value of *Value.ID* = *DID*. After that, we can perform the search in the document with identifier *Value.ID*. The *Equalize* procedure is described in [11].

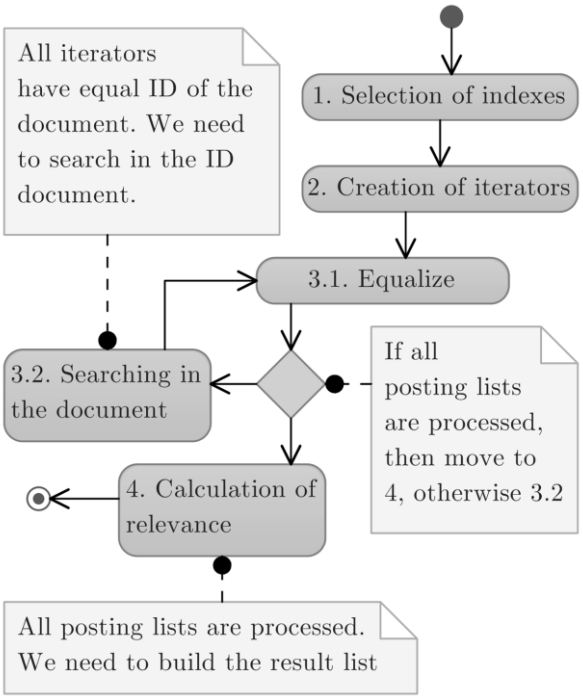

Figure 3 UML diagram of a subquery evaluation.

3.3 Index selection

To evaluate the subquery, we need to select keys for the three-component key indexes.

The query can be divided into a set of three-component keys. Let the first three lemmas of the query define the first key. Let the next three lemmas of the query define the second key, and so on.

For the cases when the length of the query is not an exact multiple of 3, the last key is always defined by the last three lemmas of the query.

All selected keys must be normalized.

For example, let us consider the subquery [who] [are] [you] [who]. We can use keys (who, are, you) and (are*, you*, who).

For any three stop lemmas *f*, *s* and *t*, we have the (*f*, *s*, *t*) index only for the case in which $f \le s \le t$. We call the (*f*, *s*, *t*) key with the aforementioned condition the normalized key.

The normalized keys here are (you, are, who) and (you*, are*, who).

Let us consider the search query "Who are you and why did you say what you did" and its subquery [who] [are] [you] [and] [why] [do] [you] [say] [what] [you] [do].

In fact, we can find this query in Cecil Forester Scott's novel "Lord Hornblower".

We can use the (who, are, you), (and, why, do), (you, say, what), and (what*, you, do) indexes.

The normalized keys are (you, are, who), (and, do, why), (you, what, say), and (you, what*, do).

We mark "what" by "*" in the last key to denote that this lemma has already been taken into account by a previous key.

3.3 Search in the document

The algorithm of searching in the document is described in Figure 4.

Let *DID* be an argument of the "Search in the document" procedure. Let us define that *DID* is the identifier of the current document.

The main difference between the search algorithm from [11] and the new search algorithm is described here.

For any lemma of the search query, we create an intermediate list of postings in memory.

For example, let us consider the three-component index (you, are, who) and its iterator object.

We create three intermediate posting lists: *IL*(you), *IL*(are), and *IL*(who).

To fill these intermediate posting lists, we need to read postings from the (you, are, who) iterator object.

A record from the (you, are, who) iterator object has the format (*ID, P, D1*, *D2*), where *ID* is the identifier of the document, *P* is the position of "you" in the document, *D1* is the distance from "are" to "you" in the text, and *D2* is the distance from "who" to "you" in the text.

If the lemma "are" occurs in the text after the lemma "you", then D1 > 0; otherwise, D1 < 0.

If the lemma "who" occurs in the text after the lemma "you", then D2 > 0; otherwise, D2 < 0.

We need to read from the iterator object all records with *ID* = *DID*.

We can produce three records from the (*ID, P, D1*, *D2*) record.

We need to store the (*P*) record in the *IL*(you) intermediate posting list.

We need to store the (*P* + *D1*) record in the *IL*(are) intermediate posting list.

We need to store the *(P* + *D2)* record in the *IL*(who) intermediate posting list.

Let us consider the key (you, what*, do) with a lemma marked by "*". In this case, we create only two intermediate posting lists, namely, *IL*(you) and *IL*(do). The (what*) component is already taken into account by a previous key.

Since for each lemma of the subquery, we have the intermediate posting list, the search is straightforward and similar to the search in the ordinary inverted file.

We can also say that an intermediate posting list is a kind of iterator object. The intermediate posting list object *IL* has the method *IL.Next*, which reads the next record from the posting list.

The intermediate posting list object *IL* has the property *IL.Value* that contains the current record (*P*), where *P* is the position of its lemma in the document.

Let *IL.Value* be equal to *SIZE_MAX* when all records are read from the *IL* object, where *SIZE_MAX* is some large number.

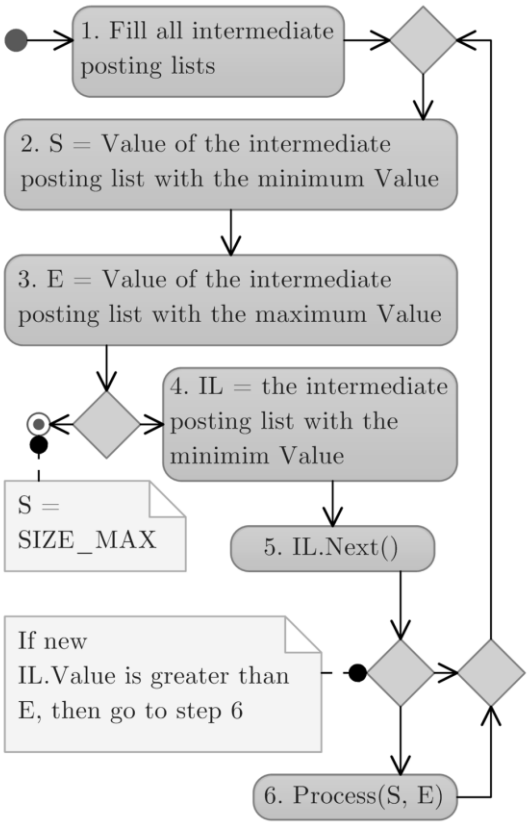

Figure 4 UML diagram of searching in the document.

In the loop, we perform the following steps.

1. Let *MinIL* be the intermediate posting list with the minimum value of *Value*. Let *S* = *MinIL.Value*.
2. Let *MaxIL* be the intermediate posting list with the maximum value of *Value*. Let *E* = *MaxIL.Value.*
3. If there are no more records in *MinIL*, then exit from the search.
4. Execute *MinIL.Next*().
5. If *MinIL.Value* > *E*, then execute *Process*(*S, E*).
6. Go to step 1.

The *Process*(*S, E*) procedure adds the (*DID, S, E*) record into the result set. *S* is the position of the start of the fragment of text within the document that contains the query. *E* is the position of the end of the fragment of text within the document that contains the query.

3.4 Intermediate posting list data ordering

The records (*P*) must be stored in an intermediate posting list for the given lemma in increasing order.

From this requirement, the following problem arises.

Consider the text "to be or not to be or".

Let the position of a word in the text be its ordinal number starting with zero.

When we create the three-component key index, the following records must be stored for the key (to, be, or). The records in the format (position of "to", position of "be", position of "or") are presented below.

(to, be, or): (0, 1, 2), (0, 5, 6), (4, 1, 2), and (4, 5, 6). From this posting list, we can create the following three intermediate posting lists.

(to): 0, 0, 4, 4; (be): 1, 5, 1, 5; and (or): 2, 6, 2, 6.

Only for the first component of the key is the intermediate posting list ordered in increasing order.

Please note that the postings in the three-component key index will actually be encoded in the (*ID*, *P*, *D1*, *D2*) format. For the (to, be, or) key, we will write the following posting list.

(to, be, or): (*ID*, 0, 1, 2), (*ID*, 0, 5, 6), (*ID*, 4, –3, –2), (*ID*, 4, 1, 2).

To solve the aforementioned problem, we create two binary heaps [17]. The first binary heap we create for the second component of the key. The second binary heap we create for the third component of the key.

Therefore, we will create the (be) binary heap and the (or) binary heap.

We limit the binary heap length by *MaxDistance* × 2.

When we need to read postings from the (to, be, or) posting list, we do the following in a loop.

1. Read the next posting (*ID*, *P*, *D1*, *D2*) from the posting list (to, be, or).
2. Write (*P*) into the (to) intermediate posting list.
3. Write (*P* + *D1*) into the (be) binary heap.
4. If the length of the (be) binary heap > *MaxDistance* × 2, then remove the first element (the minimum element) from this binary heap and write it into the (be) intermediate posting list.
5. Write (*P* + *D2*) into the (or) binary heap.
6. If the length of the (or) binary heap > *MaxDistance* × 2, then remove the first element (the minimum element) from this binary heap and write it into the (or) intermediate posting list.
7. Go to step 1.

Let us consider a key (*f*, *s*, *t*) and its posting list *L*. We create three intermediate posting lists and two binary heaps to proceed as follows.

1. Intermediate posting list *F* for *f*.
2. Intermediate posting list *S* and binary heap *SH* for *s*.
3. Intermediate posting list *T* and binary heap *TH* for *t*.

Let us introduce the methods *PopMin* and *Length* of a binary heap object. The *PopMin* method returns the minimum element from the binary heap and removes this element from the binary heap. The *Length* method returns the length of the binary heap.

In Figure 5, we present the UML diagram of the posting list *L* reading process.

After all postings from *L* are read, we need to write all elements from the binary heaps to their intermediate posting lists.

3.5 Advantages of the new algorithm

The new algorithm may require a smaller amount of posting lists to evaluate a search query in comparison with the algorithm from [11].

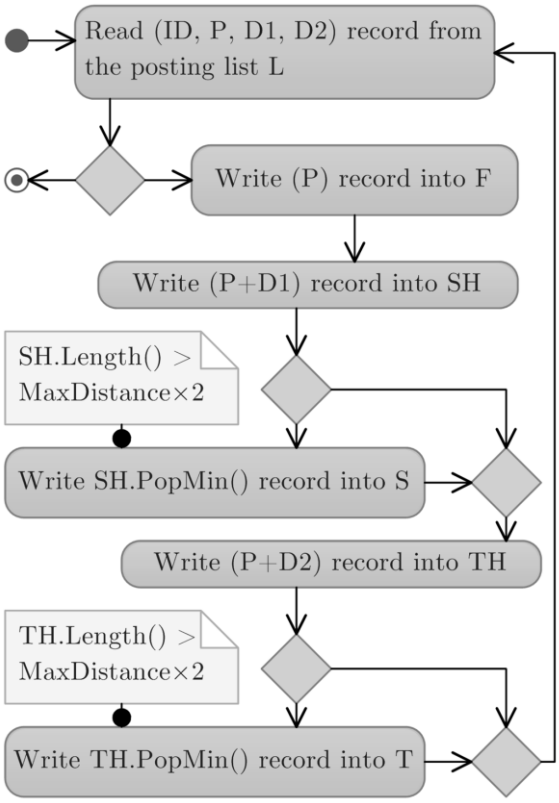

Figure 5 UML diagram of the posting list reading process.

3.6 Computational Complexity

Let *Q* be a subquery of *m* lemmas. Let *n* be the total number of postings to read for the query evaluation.

For each posting, we need to use it in the *Equalize* procedure. In [11], the author states that the cost of such usage is $O(\log(m))$.

For each posting, we need to add it to the three intermediate posting lists. The cost of this process is $O(\log(MaxDistance))$ (see 3.4).

For each posting, we need to use it when searching in a document. The cost of this process is $O(\log(m))$ (see 3.3).

The final cost of the subquery evaluation is

$O(n \cdot (\log(m) + \log(MaxDistance)) =$

$O(n \cdot \log(\max(m, MaxDistance))$.

4 Search experiments

4.1 Search experiment environment

All search experiments were conducted using a collection of texts from [11]. The total size of the text collection is 71.5 GB. The text collection consists of 195 000 documents of plain text, fiction and magazine articles. We use *MaxDistance* = 5, *SWCount* = 700, and *FUCount* = 2100. The search experiments were conducted using the experimental methodology from [11].

We assume that in typical texts, words are distributed similarly, in accordance with Zipf's law [20]. Therefore, the results obtained with our text collection will be

relevant to other collections.

We used the following computational resources:

CPU: Intel(R) Core(TM) i7 CPU 920 @ 2.67 GHz.

HDD: 7200 RPM. RAM: 24 GB.

OS: Microsoft Windows 2008 R2 Enterprise.

We created the following indexes.

Idx1: the ordinary inverted index without any improvements, such as NSW records [11, 14]. The total size is 95 GB.

Idx2: our indexes, including the ordinary inverted index with the NSW records and the (*w*, *v*) and (*f*, *s*, *t*) indexes, where *MaxDistance* = 5. The total size is 746 GB.

Please note that the total size of each type of index includes the size of the repository (indexed texts in compressed form), which is 47.2 GB.

4.2 Search results

There are 975 queries, and all queries consisted only of stop lemmas. The query set was selected as in [11]. All searches were performed in a single program thread. We searched all queries from the query set with different types of indexes to estimate the performance gains of our indexes.

The query length was from 3 to 5 words.

Studies by Spink et al. [5] have shown that queries with lengths greater than 5 are very rare. In [5], query logs of a search system were analyzed, and it was established that queries with a length of 6 represent approximately 1% of all queries and that fewer than 4% of all queries had more than 6 terms.

We performed the following experiments.

SE1: all queries are evaluated using the standard inverted index Idx1.

SE2.1: all queries are evaluated using Idx2 and the algorithm from [11].

SE2.2: all queries are evaluated using Idx2 and the new algorithm presented in this paper.

Average query times:

SE1: 31.27 sec., SE2.1: 0.33 sec., and SE2.2: 0.29 sec.

Average data read sizes per query:

SE1: 745 MB, SE2.1: 8.45 MB, and SE2.2: 6.82 MB.

Average numbers of postings per query:

SE1: 193 million, SE2.1: 765 thousand, and SE2.2: 559 thousand.

We improved the query processing time by a factor of 94.7 with the SE2.1 algorithm and by a factor of 107.8 with the SE2.2 algorithm (see Figure 6).

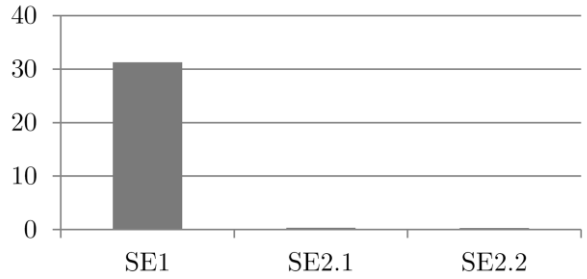

Figure 6 Average query execution times for SE1, SE2.1 and SE2.2 (seconds).

The left-hand bar shows the average query execution time with the standard inverted indexes. The subsequent bars show the average query execution times with our indexes with the SE2.1 and SE2.2 algorithms. Our bars are much smaller than the left-hand bar because our searches are very quick.

We improved the data read size per query by a factor of 88 with SE2.1 and by a factor of 109.2 with SE2.2 (see Figure 7).

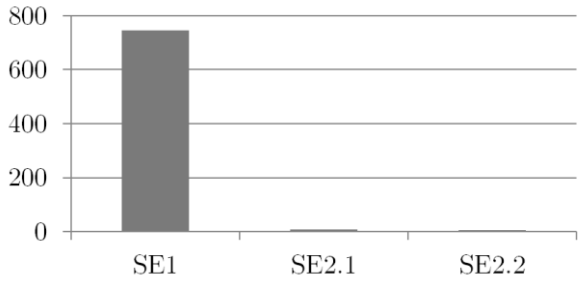

Figure 7 Average data read sizes per query for SE1, SE2.1, and SE2.2 (MB).

The left-hand bar shows the average data read size per query with SE1. The subsequent bars show the average data read size per query with SE2.1 and SE2.2.

4.3 Comparison between three-component key indexes and two-component key indexes.

We created another additional index especially for this experiment.

Idx3: two-component key indexes (*w*, *v*), where *MaxDistance* = 5, *SWCount = 0*, and *FUCount = 700*.

The total index size is 275 GB.

In this case, for any two lemmas *w* and *v*, where $w \le v$, $FL(w) < 700$, and $FL(v) < 700$, we have a two-component key index (*w*, *v*).

Each posting in this index includes the distance between *w* and *v* in the text.

Such *w* and *v* lemmas are stop lemmas for Idx2.

We performed the following experiment:

SE3: all 975 aforementioned queries are evaluated using Idx3 and the new algorithm presented in this paper is adapted for two-component key indexes.

We processed in SE3 the same query set that we already processed in SE2.1 and SE2.2 but used two-component key indexes instead of three-component key indexes.

Average query times: SE3: 3.75 sec. (see Figure 8).

Average data read sizes per query: SE3: 105.17 MB.

Average number of postings per query:

SE3: 12 million 761 thousand.

In this experiment, we compared SE2.1 and SE2.2 against SE3. We improved the query processing time by a factor of 11.36 with the SE2.1 algorithm and by a factor of 12.93 with the SE2.2 algorithm in comparison with the two-component key index (SE3) case (see Figure 8).

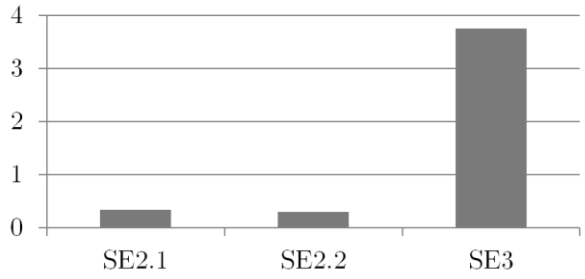

Figure 8 Average query execution times for SE2.1, SE2.2 and SE3 (seconds).

The left-hand bar shows the average query execution time with the three-component key indexes using the algorithm from [11]. The center bar shows the average query execution time with the three-component key indexes using the new algorithm from this paper. The right-hand bar shows the average query execution time with the two-component key indexes.

The bars that related to the three-component key indexes are much smaller than the right-hand bar because the three-component key indexes enable much quicker searches than the two-component key indexes.

This experiment shows that three-component key indexes **by an order of magnitude are more effective** than the two-component indexes when the queries that consist of stop lemmas are evaluated.

We improved the data read size per query by a factor of 12.44 with SE2.1 and by a factor of 15.42 with SE2.2 in comparison with the two-component key index (SE3) case (see Figure 9).

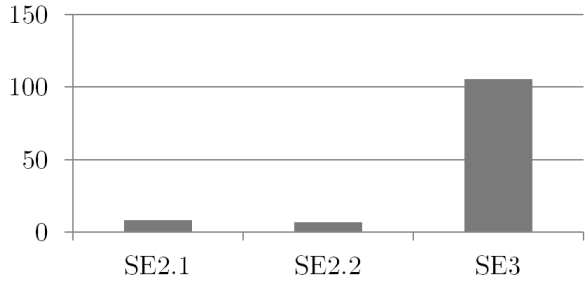

Figure 9 Average data read sizes per query for SE2.1, SE2.2, and SE3 (MB).

The left-hand bar shows the average data read size per query with SE2.1. The subsequent bars show the average data read size per query with SE2.2 and SE3.

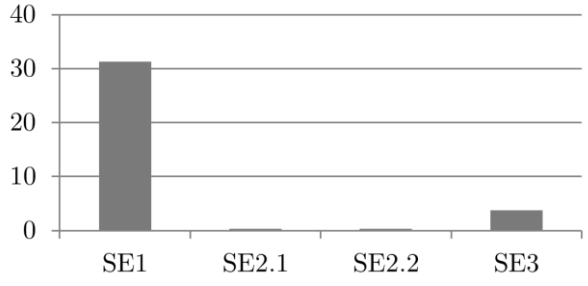

Figure 10 Average query execution times for SE1, SE2.1, SE2.2 and SE3 (seconds).

We show the average query execution time for all experiments in Figure 10.

The left-hand bar shows the average query execution time with the standard inverted indexes. The two subsequent bars show the average query execution times with the three-component key indexes with the SE2.1 and SE2.2 algorithms. The right-hand bar shows the average query execution time with the two-component key indexes in experiment SE3.

The author decided not to use logarithmic scales in the diagrams because users do not think in logarithmic scales when evaluating the search speed of a search system.

5 Conclusion and future work

A query that contains high-frequency occurring words induces performance problems. To solve these performance problems and to satisfy the fastidious demands of the users, we developed and elaborated three-component key indexes.

In this paper, we investigated searches with queries that contain only stop lemmas. Other query types are studied in [13, 14].

As we discussed in [11], three-component key indexes are an important and integral part of our comprehensive full-text search methodology, which comprises three-component key index search methods and other search methods from [13, 14].

In this paper, we have introduced an optimized algorithm for full-text searches in comparison with [11]. These algorithms are novel, and no alternative implementations exist.

We have presented the results of experiments showing that when queries contain only stop lemmas, the average time of the query execution with our indexes is 107.8 times less (with the value of *MaxDistance* = 5) than that required when using ordinary inverted indexes.

We have presented the results of experiments showing that when queries contain only stop lemmas, the average time of the query execution with our indexes is 12.93 times less (with the value of *MaxDistance* = 5) than that required when using two-component key indexes.

Using the last experiment, we diligently prove that three-component indexes are stupendous and cannot be replaced by two-component key indexes. This is the reason why we implemented three-component indexes to solve the full-text search task.

In the future, it will be interesting to investigate other types of queries in more detail and to optimize index creation algorithms for larger values of *MaxDistance*. It will also be important to investigate how the proposed indexing structure can be used by modern ranking algorithms. The author assumes that based on Zipf's law [20], our test text collection is sufficient and acceptable for evaluating the search performance. Nevertheless, to investigate ranking algorithms' behavior we plan to use collections such as TREC GOV and GOV2, which are intended to analyze the search quality.

References

[1] V.N. Anh, O. de Kretser, A. Moffat: Vector-Space Ranking with Effective Early Termination. In: SIGIR '01 Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New Orleans, Louisiana, USA, September 9–12, 2001, pp. 35–42.

[2] S. Buttcher, C. Clarke, B. Lushman: Term proximity scoring for ad-hoc retrieval on very large text collections. In: SIGIR'2006, pp. 621–622.

[3] D. Bahle, H.E. Williams, J. Zobel. Efficient Phrase Querying with an Auxiliary Index. In: SIGIR '02 Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Tampere, Finland, August 11–15, 2002, pp. 215–221.

[4] S. Garcia, H.E. Williams, A. Cannane: Access-Ordered Indexes. In: ACSC '04 Proceedings of the 27th Australasian Conference on Computer Science, Dunedin, New Zealand, January 18–22, 2004, pp. 7–14,.

[5] B.J. Jansen, A. Spink, T. Saracevic: Real life, real users and real needs: A study and analysis of user queries on the Web. Information Processing and Management, 36(2): 207–227 (2000).

[6] R. W. P. Luk: Scalable, statistical storage allocation for extensible inverted file construction. Journal of Systems and Software, 84(7): 1082-1088 (2011).

[7] R.B. Miller: Response Time in Man-Computer Conversational Transactions. In Proceedings: AFIPS Fall Joint Computer Conference. San Francisco, California, December 09-11, 1968, vol 33, pp. 267–277.

[8] Y. Rasolofo, J. Savoy: Term Proximity Scoring for Keyword-Based Retrieval Systems. In: European Conference on Information Retrieval (ECIR) 2003: Advances in Information Retrieval, pp. 207–218.

[9] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, G. Weikum: Efficient text proximity search. In: String processing and information retrieval. 14th International Symposium. SPIRE 2007. October 29–31, 2007. Lecture notes in computer science. Santiago de Chile, Chile, Springer, Berlin, Heidelberg, vol. 4726, pp. 287–299.

[10] A. Tomasic, H. Garcia-Molina, K. Shoens. Incremental updates of inverted lists for text document retrieval. SIGMOD '94 Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. Minneapolis, Minnesota, May 24–27, 1994, pp. 289–300.

[11] A.B. Veretennikov: Proximity full-text search with response time guarantee by means of three component keys. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering. 7(1): 60–77 (2018) (in Russian). doi: http://dx.doi.org/10.14529/cmse180105.

[12] A.B. Veretennikov: O poiske fraz i naborov slov v polnotekstovom indekse (About phrases search in full-text index). Control systems and information technologies, 48(2.1): 125–130 (2012) (In Russian).

[13] A.B. Veretennikov: Ispol'zovanie dopolnitel'nykh indeksov dlya bolee bystrogo polnotekstovogo poiska fraz, vklyuchayushchikh chasto vstrechayushchiesya slova (Using additional indexes for fast full-text searching phrases that contains frequently used words). Control Systems and Information Technologies, 52(2): 61–66 (2013) (In Russian).

[14] A.B. Veretennikov: Effektivnyi polnotekstovyi poisk s ispol'zovaniem dopolnitel'nykh indeksov chasto vstrechayushchikhsya slov (Efficient full-text search by means of additional indexes of frequently used words). Control Systems and Information Technologies, 66(4): 52–60 (2016) (In Russian).

[15] A.B. Veretennikov: Sozdanie dopolnitel'nykh indeksov dlya bolee bystrogo polnotekstovogo poiska fraz, vklyuchayushchikh chasto vstrechayushchiesya slova (Creating additional indexes for fast full-text searching phrases that contains frequently used words). Control systems and information technologies, 63(1): 27–33 (2016) (In Russian).

[16] A.B. Veretennikov: Effektivnyi polnotekstovyi poisk s uchetom blizosti slov pri pomoshchi trekhkomponentnykh klyuchei (Efficient full-text proximity search by means of three component keys). Control systems and information technologies, 69(3): 25–32 (2017) (In Russian).

[17] J.W.J. Williams: Algorithm 232 – Heapsort. Communications of the ACM. 7(6): 347–348, (1964).

[18] H.E. Williams, J. Zobel, D. Bahle. Fast Phrase Querying with Combined Indexes. ACM Transactions on Information Systems (TOIS), 22(4): 573–594 (2004).

[19] H. Yan, S. Shi, F. Zhang, T. Suel, J.-R. Wen: Efficient Term Proximity Search with Term-Pair Indexes. In: CIKM '10 Proceedings of the 19th ACM International Conference on Information and Knowledge Management, Toronto, ON, Canada, October 26-30, 2010, pp. 1229–1238.

[20] G. Zipf: Relative Frequency as a Determinant of Phonetic Change. Harvard Studies in Classical Philology, 40: 1–95, 1929.

[21] J. Zobel, A. Moffat: Inverted files for text search engines. ACM computing surveys, 2006, 38(2), Article 6.