UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TO INDEX OR NOT TO INDEX: TIME-SPACE TRADE-OFFS IN SEARCH ENGINES WITH POSITIONAL RANKING FUNCTIONS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

SENEN ANDRÉS GONZÁLEZ CORNEJO

PROFESOR GUÍAS:
GONZALO NAVARRO BADINO
DIEGO ARROYUELO BILLIARDI

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
DIEGO SECO NAVEIRAS

Este trabajo ha sido parcialmente financiado por Fondecyt Grant 1-110066, and Conicyt Project 78100003.

SANTIAGO DE CHILE
ABRIL 2014

Abstract

Web search has become an important part of day-to-day life. Web search engines are important tools that give access to the information stored in the web. The success of a web search engine mostly depends on its efficiency and the quality of its ranking function. But also, web search engines give extra aids to their users, which make them more usable. An instance of this is the ability of generating result snippets and being able to retrieve the in-cache version of a web page, among others. Inverted indexes are a fundamental data structure used by web search engines to efficiently answer user queries. In a basic setup, inverted indexes only allow for simple (though fairly effective) ranking functions (e.g., BM25). It is well known that the high quality of nowadays search-engine results is due to sophisticated ranking functions. A particular example that has been widely studied in the literature is that of positional ranking functions, where the positions of the query terms within the resulting documents are used in order to rank them. To support this kind of ranking, the classical solution are positional inverted indexes. However, these usually demand large amounts of extra space, typically about three times the space of an inverted index. Moreover, if the web search engine needs to produce text snippets or display a cached copy of a web page, the textual data must be also stored.

In this thesis we study time/space trade-offs for web search engines with positional ranking functions and text snippet generation. We aim to answer the question of whether positional inverted indexes are the most efficient way to store and retrieve positional data. In particular, we propose to get rid of positional data in inverted indexes, and instead obtain that information from the text collection itself. The challenge is to compress the text collection such that one can support the extraction of arbitrary documents, in order to find the positions of the query terms within them. We study and compare several alternatives for compressing the textual data. The first one uses a succinct data structure (in particular, a Wavelet Tree). We show how the space of the data structure can be reduced significantly, but also slowed down, by using high-order compressors within the nodes of the data structure. We then show how several text compression alternatives behave when used to obtain arbitrary documents (note that decompression speed is key in this application). Our starting point are compressors that either: (1) use little space for the text, yet with a slow decompression speed; and (2) have a very efficient decompression time (achieving a total performance comparable to that of positional inverted indexes), yet with a poor compression ratio. We then show how to obtain the best from both worlds: an efficient compression ratio, with a high decompression speed.

We conclude that there exist a wide range of practical time/space trade-offs, other than just positional inverted indexes. The main result is that using only about 50% of the space of current solutions (i.e., positional inverted indexes plus the compressed text), one can support positional ranking and snippet generation almost with no time penalties. This seems to indicate that "not to index" positional data is the best solution in many practical scenarios. This can change the way in which positional data is stored and retrieved in web search engines.

Dedicado a mi Esposa Carmen y a mi hija Jazmín,
pues son la felicidad de mi vida.

Thanks

Quisiera partir agradeciendo a mi esposa y mi hija por darme todo su apoyo, y a mis padres por llevarme adelante. Al profesor Gonzalo Navarro, por aceptarme como alumno. A Diego Arroyuelo quien me guió en el proceso del trabajo. Al laboratorio Yahoo! Reseach chile, por facilitarme las herramientas para poder realizar este trabajo. A mi segunda familia CROSS, por estar siempre conmigo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Web search has become an important part of day-to-day life, affecting even the way in which people think and remember things [43]. Indeed, *web search engines* are one of the most important tools that give access to the huge amount of information stored in the web. The success of a web search engine mostly depends on its efficiency and the quality of its ranking function. To achieve efficient processing of queries, search engines use highly optimized data structures, including inverted indexes [8, 13, 29]. State-of-the-art ranking functions, on the other hand, combine simple term-based ranking schemes such as BM25 [13], link-based methods such as Pagerank [9] or Hits [28], and up to several hundred other features derived from documents and search query logs.

Recent work has focused on *positional ranking functions* [38, 32, 44, 33, 40, 49, 13] that improve result quality by considering the positions of the query terms in the documents. Thus, documents where the query terms occur close to each other might be ranked higher, as this could indicate that the document is highly relevant for the query. To support such positional ranking, the search engine must have access to the position data. This is commonly done by building an index for all term positions within documents, called a *positional index*. The goal is to obtain an index that is efficient in terms of both index size and access time.

As shown in [38], positional ranking can be carried out in two phases. First, a simple term-based ranking scheme (such as BM25) defined over a Boolean filter is used to determine a set of high-scoring documents, say, the top 200 documents. In the second phase, the term positions are used to rerank these documents by refining their score values. (Additional third or fourth phases may be used to do further reranking according to hundreds of additional features [46], but this is orthogonal to our work.) Once the final set of top-scoring documents has been determined (say, the top 10), it is necessary to generate appropriate *text snippets*, typically text surrounding the term occurrences, to return as part of the result page. This requires access to the actual text in the indexed web pages. It is well known [49, 25] that storing position data requires a considerable amount of space, typically about 3 to 5 times the space of an inverted index storing only document identifiers and term frequencies. Furthermore, storing the documents for snippet generation requires significant additional space.

1.2 Positional Indexes

Compressing positional data is a problem where it has been difficult to make much progress. For instance, previous work [49] concludes that term positions in the documents do not follow simple distributions that could be used to improve compression (as is the case of, for instance, `docID`s and frequencies). As a result, a positional index is about 3 to 5 times larger than a `docID`/frequency index, and becomes a bottleneck in index compression. Another important conclusion from [49] is that we may only have to access a limited amount of position data per query, and thus it might be preferable to use a method that compresses well even if its speed is slightly lower. Positions in inverted indexes are used mainly in two applications, phrase searching [29] and positional ranking schemes. In this work we focus in the use of positions to improve the ranking, where the positions of the query terms within the documents are used to enhance the performance of a standard ranking such as BM25. The rationale is that documents where the query terms appear close together could be more relevant for the query, so they should get a better score. Although in this work we focus only on the use of positions to improve ranking, the compression schemes used allow for phrase searching as well. This scenario is left for future work. A recent work on positional indexing is that of Shan et al. [42]. They propose to use the *flat position indexes* [15, 18] as an alternative of positional inverted indexes. The result is that `docID`s, term frequencies and position data can be stored in space close to that of positional inverted lists, yielding a reduction of space usage. However, this index does not store the text, which makes it less suitable in scenarios where text snippets must be generated.

1.3 Hypothesis

In this thesis we prove the following hypothesis:

> Position data in web indexes can be obtained more efficiently (in terms of processing time and space usage) using compressed text representations rather than the classical solution of positional inverted indexes.

1.4 Thesis contribution

In this thesis we study what are the best ways to organize and store the in-document positions and textual data, in order to efficiently support positional ranking and snippet generation in text search engines. One of our main conclusions is that some compressed representations of the textual data — which are needed to support snippet generation — can also be used to efficiently obtain the term positions needed by positional ranking methods. Therefore, *no positional index is needed in many cases*, thus saving considerable space at little cost in terms of running time.

Our main contributions can be summarized as follows:

1. A study of several trade-offs for compressing position data. Rather than storing a positional index, we propose to compute the term positions from a compressed representation of the text. This is shown in Chapters 4 and 5, following the description explained in Section 3.4. We explore and propose several compression alternatives. Our results significantly enhance current trade-offs between running time and memory space usage, enabling in this way more design choices for web search engines. One of our most interesting results is that both position and textual data can be stored in about 66% the space of current positional inverted indexes.
2. A study of several alternatives for compressing textual data, extending the alternatives studied in previous work [25]. In particular, we show that using the scheme in [45] (to compress text and support efficient snippet generation) before using a standard compressor yields good time-space trade-offs, extending the alternatives introduced in [22]. This study is explained in details in Chapter 5. It is important to note that variants of the scheme in [45] have been adopted by some commercial search engines, which makes our results of practical interest.
3. We propose several improvements over wavelet trees [27], in order to make them competitive for representing document collections. The details of the improvement are explained in Chapter 4. We show how to improve the compression ratio by compressing the sequence associated to every node of the tree with standard compressors. The result is a practical web-scale compressed self-index that is competitive with the best state-of-the-art compressors.

1.5 Results

The results of this work have been partially published in:

Diego Arroyuelo, Senén González, Mauricio Marín, Mauricio Oyarzún and Torsten Suel. To Index or not to Index: Time-Space Trade-offs in Search Engines with Positional Ranking Functions. In Proc. of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval. Pages 255–264. Portland, OR, USA, August 12-16, 2012.

Chapter 2

Background and Related Work

2.1 Basic Definitions

Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a document collection of size N, where each document is represented as a sequence $D_{\mathrm{i}}[1..n_{\mathrm{i}}]$ of n_{i} terms from a vocabulary $\Sigma = \{1, \ldots, V\}$. Notice that every term is represented by an integer, hence the documents are just arrays of integers.

Definition 2.1 Given a document D_{i}, we identify it with the unique document identifier (`docID`) i.

Definition 2.2 Given a term $t \in \Sigma$ and a document $D_{\mathrm{i}} \in \mathcal{D}$, the *in-document positions* of t in D_{i} is the set $\{j | D_{\mathrm{i}}[j] = t\}$.

2.2 Text Representation

Throughout this work, we assume that all term separators (like spaces, '`,`', '`;`', '`.`', etc.) have been removed from the text. Also, we assume that all terms in the vocabulary have been represented in a case-insensitive way. This is in order to facilitate the search operations that we need to carry out over the documents in order to compute (on the fly) the positions of a given query term.

To be able to retrieve the original text (with separators and the original case) one can use the *presentation layer* introduced by Fariña et al. [21, Section 4]. This also supports removing stopwords and the use of stemming, among other vocabulary techniques. This extra layer requires extra space on top of that of the compressed text, as well as extra time to obtain the original text. However, this scheme must be used on all the alternatives that we consider in this work, and thus we disregard the overhead introduced by the presentation layer and focus only on the low-level details of compression (but keeping in mind that the original text can still be retrieved).

2.3 Inverted Indexes

The efficiency of query processing in search engines relies on *inverted indexes*. These data structures store a set of *inverted lists* $I_1, \ldots, I_V$, which are accessed through a *vocabulary table*.

Example 2.1 An example of such data structure can be seen in Figure 2.1, for a document collection of 4 documents $\{D_1, D_2, D_3, D_4\}$.

The list I_t maintains a *posting* for each document containing the term $t \in \Sigma$. Usually, a posting in an inverted list consists of a `docID` a term frequency, and the in-document positions of the term.

Example 2.2 An example can be seen in Figure 2.2, where the inverted list of a term t_3 is shown. Each posting on the list stores a `docID` the frequency, and the in-document positions.

In real systems, the `docID`s, term frequencies and in-document positions are often stored separately. Indexes whose postings store in-document positions are called *positional inverted indexes*. The inverted lists of the query terms are used to produce the result for the query. Since the query results are usually large, the result set must be ranked by relevance.

2.4 Inverted Index Compression

For large document collections, the data stored in inverted indexes requires considerable amounts of space. Due to the large volume of data on the web and its rapid growth, data compression has become important [48, 39]. In the case of inverted indexes for web search engines, this effect is reflected in the high space usage of the resulting inverted lists. Hence, the indexes must be compressed. To support efficient query processing (such as DAAT [13], WAND [12] or BMW OR [19]) and effective compression in the inverted lists, we sort them by increasing `docID`.

Also, to avoid decompressing whole lists at query time, we assume that an inverted list I_t is divided into chunks of 128 documents each. The particular choice of 128 documents per chunk is an implementation issue. Given a chunk of list I_t, the term-position data for all the documents in that chunk are stored in a *separate* chunk of variable size.

Let $\mathrm{d}_t[1..|I_t|]$ denote the sorted list of `docID`s for the inverted list I_t. Then, we replace $\mathrm{d}_t[1]$ with $\mathrm{d}_t[1] - 1$, and $\mathrm{d}_t[\mathrm{i}]$ with $\mathrm{d}_t[\mathrm{i}] - \mathrm{d}_t[\mathrm{i} - 1] - 1$ for $\mathrm{i} = 2, \ldots, |I_t|$. In the

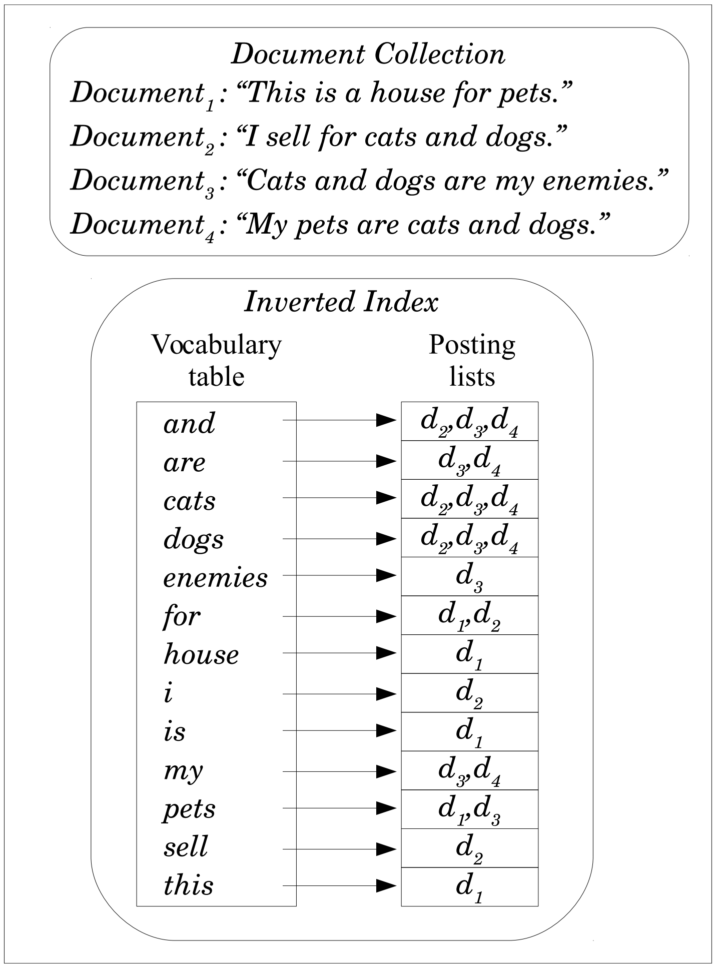


Figure 2.1: Example of an inverted index for a document collection, storing just `docIDs`.

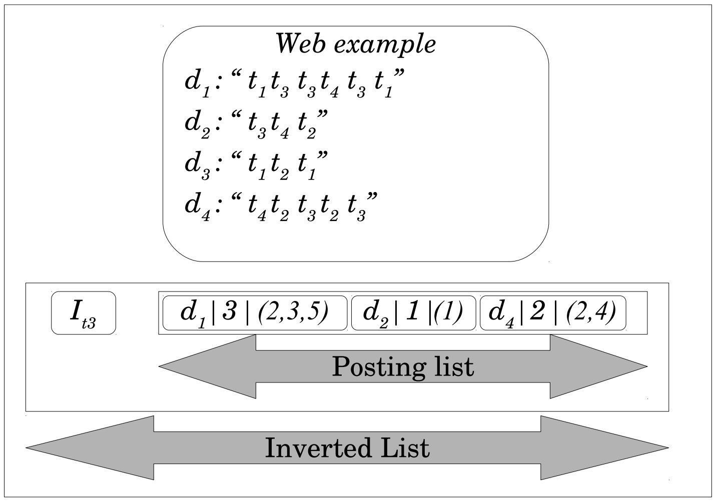


Figure 2.2: Example of an inverted list for the term t_3.

case of frequencies, every f_{i} is replaced with $f_{\mathrm{i}} - 1$, since $f_{\mathrm{i}} > 0$ always holds. For the positions, each $p_{\mathrm{i},j}$ is replaced with $p_{\mathrm{i},j} - p_{\mathrm{i},j-1} - 1$. Then these values are encoded with integer compression schemes that take advantage of the resulting smaller integers.

There has been a lot of progress on compressing `docID` and frequencies, with many compression methods available [50, 13]. Some of them achieve, in general, a very good compression ratio, but at the expense of a lower decompression speed [13], for example Elias γ and δ encodings [20], or Golomb/Rice parametric encodings [26], interpolative encoding [35]. Other methods achieve a (slightly) lower compression ratio, though with much higher decompression speed, for example *VByte* [47], *S9* [4], and `PforDelta` [54], among others [13]. The best compression method depends on the scenario at hand.

To achieve compression, some of the following special features of posting lists are used:

- `docID` represent documents that are related to the term, so these are positive integers.
- There are no repeated elements in the lists.
- The lists are sorted by increasing `docID`, and hence the lists are represented differentially.

We review next the integer compression schemes that will be used along this thesis.

2.4.1 VByte

This method [41] represents an integer using a variable number of bytes, compressing each number separately. To do this, *VByte* use the most significant bit in a byte as a flag. This flag indicates whether the next byte corresponds to the next integer in the segment, or not. So for every byte, just 7 bits are used to represent the number. This is shown in Figure 2.3.

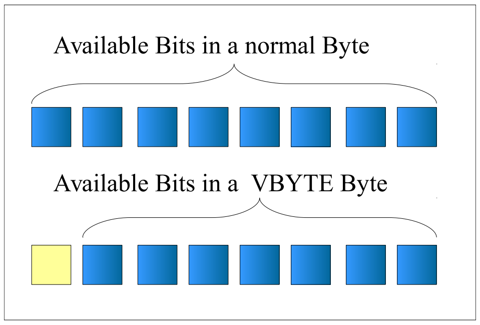

Available Bits in a normal Byte
Available Bits in a VBYTE Byte

Figure 2.3: Byte layout of *VByte*.

This flag is called continuator. If the continuator is “0” it means that the current byte is not enough to encode the current integer and hence the encoding must use the next byte. If the continuator is “1” then the current byte is the last used by the encoding. The entire process of coding and decoding is shown below.

Example 2.3 Figure 2.4 shows the 32-bit representation of the integer 167.

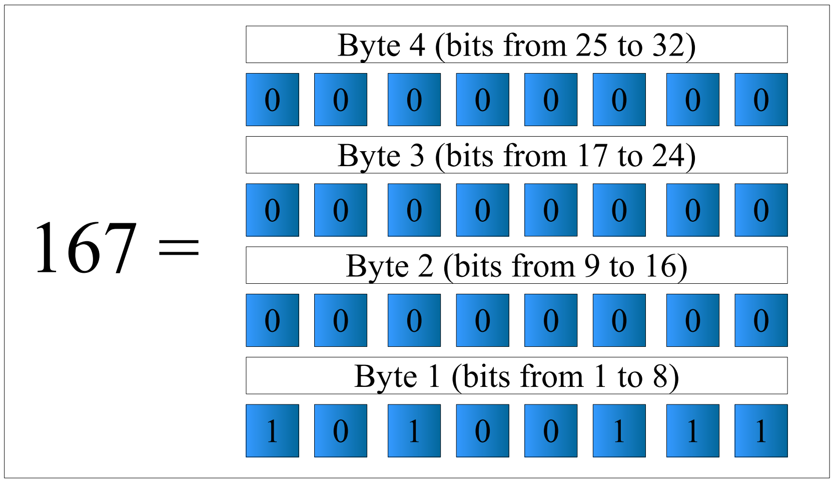

167 =
Byte 4 (bits from 25 to 32)
0 0 0 0 0 0 0 0
Byte 3 (bits from 17 to 24)
0 0 0 0 0 0 0 0
Byte 2 (bits from 9 to 16)
0 0 0 0 0 0 0 0
Byte 1 (bits from 1 to 8)
1 0 1 0 0 1 1 1

Figure 2.4: The 32-bit representation of 167.

Notice that the most-significant bytes 4, 3 and 2 are not used, so that 167 could be represented with less than 4 bytes.

To encode a number in *VByte*, we try to accommodate its binary encoding in 7 bits. If so, we store the encoding in 7 bits and mark the continuator with a “1”. Otherwise the least significant 7 bits are stored in a byte with continuator “0” and proceed with the remaining bits, until no extra bytes are needed to store the encoding.

Example 2.4 Figure 2.5 shows the encoding of 167 using *VByte*.

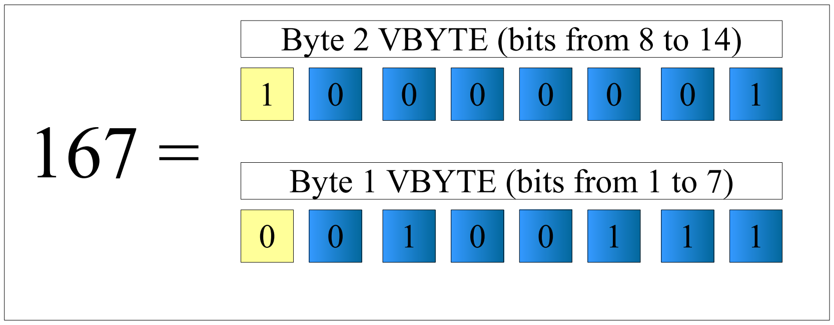

167 =
Byte 2 VBYTE (bits from 8 to 14)
1 0 0 0 0 0 0 1
Byte 1 VBYTE (bits from 1 to 7)
0 0 1 0 0 1 1 1

Figure 2.5: The encoding of the integer 167 in VBYTE.

The decoding process can be carried out very efficiently using bit-wise and arithmetic operations, as it can be seen in Figure 2.6.

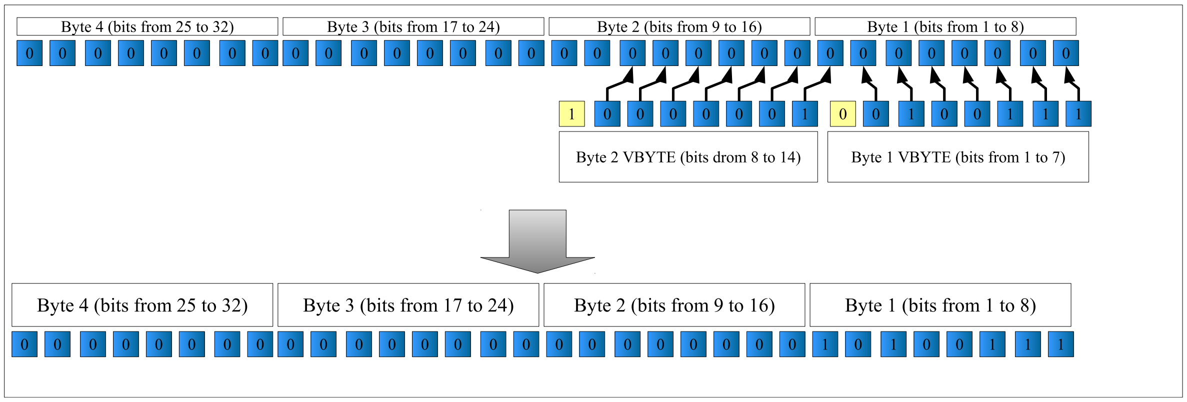
Example 2.5 The decoding process for 167 is illustrated in Figure 2.7.

Table 2.1 shows the numbers of bytes used in *VByte* for different ranges of integers.

In the best case, for every number we use 1 byte, using a 1/4 of the original space (for 32-bit integers).

```
int decodeVByte (char * compressed) {
  int v = 0;
  int i = 0;
  while (compressed[i] & '128' == '0') {
    v = v || compressed[i];
    v = v<<7;
    i++;
  }
  v = v || compressed[i];
  return v;
}
```

Figure 2.6: Decoding process for *VByte*.



Byte 4 (bits from 25 to 32)
Byte 3 (bits from 17 to 24)
Byte 2 (bits from 9 to 16)
Byte 1 (bits from 1 to 8)
Byte 2 VBYTE (bits drom 8 to 14)
Byte 1 VBYTE (bits from 1 to 7)
Byte 4 (bits from 25 to 32)
Byte 3 (bits from 17 to 24)
Byte 2 (bits from 9 to 16)
Byte 1 (bits from 1 to 8)

Figure 2.7: Decoding process for integer 167.

Table 2.1: Number of bytes used to represent an integer number with *VByte*.

Integer range	Number of bytes used
from 0 to 127	1
from 128 to 16383	2
from 16384 to 2097151	3
from 2097152 to 268435455	4
from 268435456 to 42949672952	5

The main features of this method are:

- It is simple to encode and decode integers.
- Uses at most 1 byte more than the minimum amount of bytes to represent a number.
- Uses at least 1 byte per integer encoded.

When the integers to encode are small, using 1 byte is wasteful. An improvement to this is called *variable Nibble* (*VNibble*), which is similar to *VByte* but it works at the nibble level (4 bits). The most significant bit of each nibble is used as the continuator. This can be seen in the Figure 2.8.

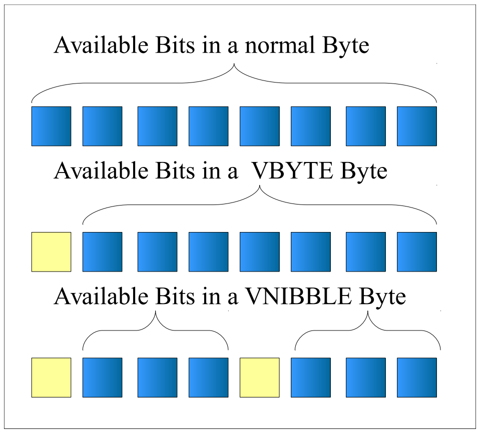

Available Bits in a normal Byte
Available Bits in a VBYTE Byte
Available Bits in a VNIBBLE Byte

Figure 2.8: Example of *VNibble* bytes.

The encoding and decoding processes are the same (or similar) than that of *VByte*. Table 2.2 shows the number of bytes used for *VNibble* for different range of integers. When the integers to encode are smaller or equal than 7, this method uses just 4 bits to represent it, which is better.

Table 2.2: Number of nibbles used to represent an integer number with *VNibble*.

Integer range	Number of nibbles used
from 0 to 7	1
from 8 to63	2
from 64 to 511	3
from 512 to 4095	4
from 4096 to 32767	5
from 32768 to 262143	6
from 262144 to 2097151	7
from 2097152 to 16777215	8
from 16777216 to 134217727	9
from 134217728 to 1073741823	10
from 1073741824 to 42949672952	11

2.4.2 Simple 9

The *VByte* and *VNibble* schemes encode each integer individually. This means that in the best scenario *VByte* is able to encode 4 integers within a 32-bit machine word, and at best 8 integers for *VNibble*. However, if the integers to encode are small (e.g. able to be encoded with 1 bit) we waste space.

The basic idea of *S9* [4] is to take several consecutive integers in the sequence and try to fit them in a 32-bit machine word, encoding each integer with a fixed equal-size chunk. The size of the chunk is defined as the number of bits needed to encode the largest number in the group. To do so, *S9* uses an *S9 word* of 32-bit. In each *S9 word*, four bits are used as a header to define how many bits will be used to encode the integers, and how many integers will be stored in the *S9 word*. The remaining 28 bits are used to store the information of the integers encoded. The *S9 word* layout is shown in Figure 2.9.

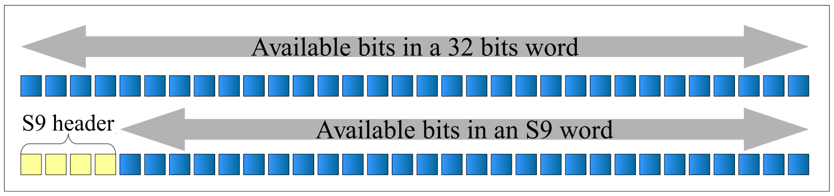

Available bits in a 32 bits word
S9 header
Available bits in an S9 word

Figure 2.9: Layout of an *S9 word*.

There are nine possible ways of dividing 28 bits into chunks of equal size (the number of cases is the reason behind the name of the method). The meaning of the cases is shown in Table 2.3.

Table 2.3: Meaning of the cases in the header of the *S9 word*.

Case	Number of integers stored	Chunk size
0	28	1
1	14	2
2	8	3
3	7	4
4	5	5
5	4	7
6	3	8
7	2	14
8	1	28

To encode integers in an *S9 word*, we try to fit the maximum amount of integers in the 28 available bits . First, we try to fit 28 integers. If that is not possible, then we try with 14, and so on, until we eventually get to the case were only one 28-bit integer can be stored. In the header of the *S9 word* we store the particular case used for it.

Example 2.6 If we encode the integers 98, 112, 117 and 121, each of these integers can encoded in binary using 7 bits, which according to Table 2.3 correspond to case 5. The resulting *S9 word* is shown in Figure 2.10.

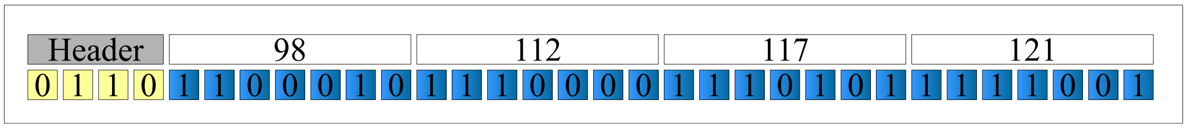


Figure 2.10: *S9 word* encoding the group of integers 98, 112, 117 and 121.

To decode an *S9 word*, first we obtain the header by means of a bit mask. Then , the case is used on a switch statement where all 9 cases have been hard-coded (as it can be seen in Figure 2.12).

Example 2.7 Figure 2.11 illustrate the decoding process for an *S9 word* that encodes the integers 98, 112, 117 and 121.

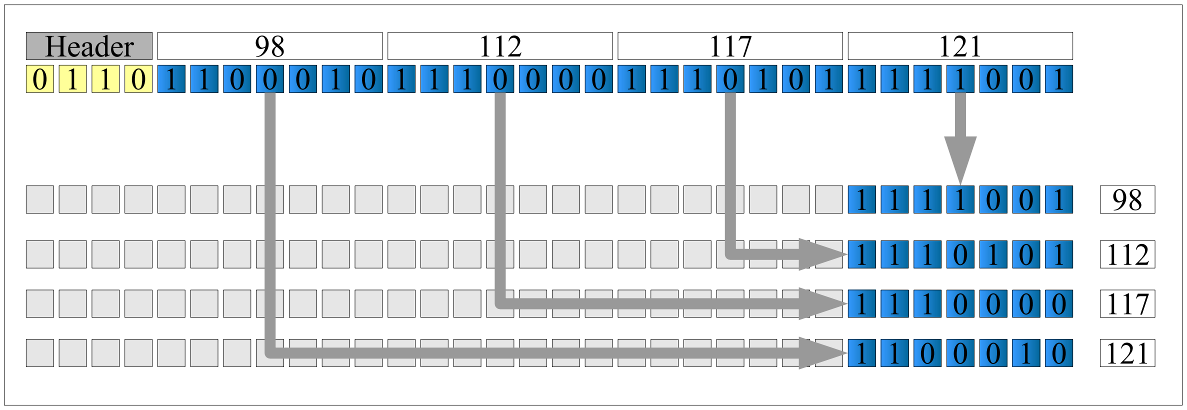


Figure 2.11: Decodification process for an *S9 word*, which encodes the group of integers $98, 112, 117$ and 121.

The main feature of this method that makes it highly efficient at decoding time is that all the integers in an *S9 word* are coded in the same amount of bits. Hence, the decoding process can be optimized by hard-coding all cases, as we already said.

An improvement of this technique is to use all the 16 possible cases that we can have with headers of 4 bits. This improvement is called S16 [51].

2.4.3 PforDelta

The `PforDelta` encoding [54] divides an inverted list into *blocks* of, usually, 128 integers each. To encode the integers within a given block, it gets rid of a given percentage—usually 10%—of the largest integers within the block, and stores them in a separate

```
void decodeS9 (int s9word, int * result) {
   char case = getHeader(s9word);
   switch(case) {
    case 0 :
    result[0] = (s9word & 0x08000000) >> 27;
    result[1] = (s9word & 0x04000000) >> 26;
    . . .
    result[26] = (s9word & 0x00000002) >> 1;
    result[27] = (s9word & 0x00000001) ;
    break;
    case 1 :
    result[0] = (s9word & 0x0c000000) >> 26;
    result[1] = (s9word & 0x03000000) >> 24;
    . . .
    result[12] = (s9word & 0x0000000c) >> 1;
    result[13] = (s9word & 0x00000003) ;
    break;
    . . .
    case 8 :
    result[0] = (s9word & 0xffffff);
    break;
   }
}
```

Figure 2.12: Decoding process for an *S9 word*.

memory space. These are the *exceptions* of the block. Next, the method finds the largest remaining integer in the block, let us say x, and represents each integer in the block in binary using $b = \lceil \log x \rceil$ bits. Though the exceptions are stored in a separate space, we still maintain the slots for them in their corresponding positions. This facilitates the decoding process. For each block we maintain a *header* that stores information about the compression used in the block, e.g., the value of b.

An illustration of a `PforDelta` block is shown in Figure 2.13. To retrieve the positions of the exceptions, we also store the position of the first exception in the header. In the slot of each exception, we store the offset to the next exception in the block. In the last exception, we store a '0'. This forms a linked list with the exception slots. In

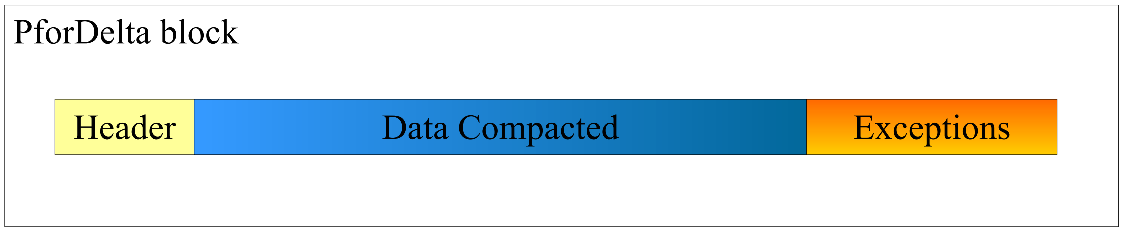


Figure 2.13: Layout of `PforDelta` block.

case that b is too small and we cannot accommodate the offset to the next exception (because the offset to the next one cannot be accommodated within b bits), the algorithm forces to add extra exceptions between two original exceptions. This increases the space usage when the lists contain many small numbers.

Example 2.8 A `PforDelta` block that is encodes 128 integers is shown in Figure 2.14. The exceptions are 78, 110, 160 and 91.

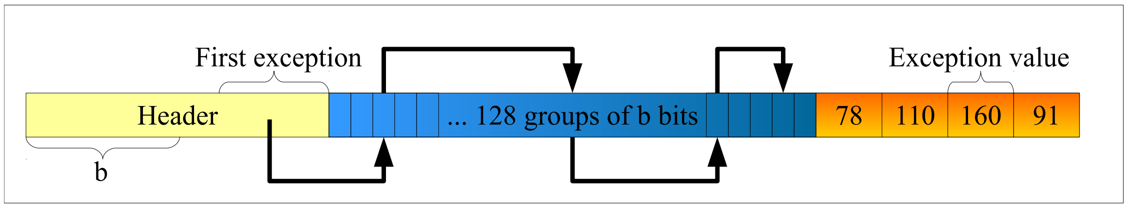


Figure 2.14: `PforDelta` block, encoding a group of integers, with exeptions $78, 110, 160$ and 91.

To decompress a block, we first take b from the header, and invoke a specialized function that obtains the b-bit integers. Each b has its own extracting function, so they can be hard-coded for high efficiency. Once we decode the integers, we traverse the list of exceptions of the block, storing the original integers in their corresponding positions. This step can be slower, yet it is carried out just for 10% of the block. The decoding process implementation is shown on Figure 2.16.

In typical implementations of `PforDelta`, the header is implemented in 32 bits, since we only need to store the values of b (in 6 bits, since $1 \leq b \leq 32$) and the position of the first exception (in 7 bits, since the block has 128 positions). `PforDelta` has shown to be among the most efficient compression schemes [50], achieving a high decompression speed.

Example 2.9 Figure 2.15 shows the size and the decoding process for the `PforDelta` block previously mentioned.

2.5 Snippet Generation

Besides providing a ranking of the most relevant documents for a query, search engines must show query snippets and support accessing the "in-cache" version of the documents. To achieve this, web search engines must store a copy of document collection.

Example 2.10 Figure 2.17 shows an example of user-query result, and the result snippets.

Each snippet shows a relevant portion of the result document, in order to help the user judge its likely relevance before accessing it. Turpin et al. [45] introduce a method to compress the text collection and support fast text extraction to generate snippets.

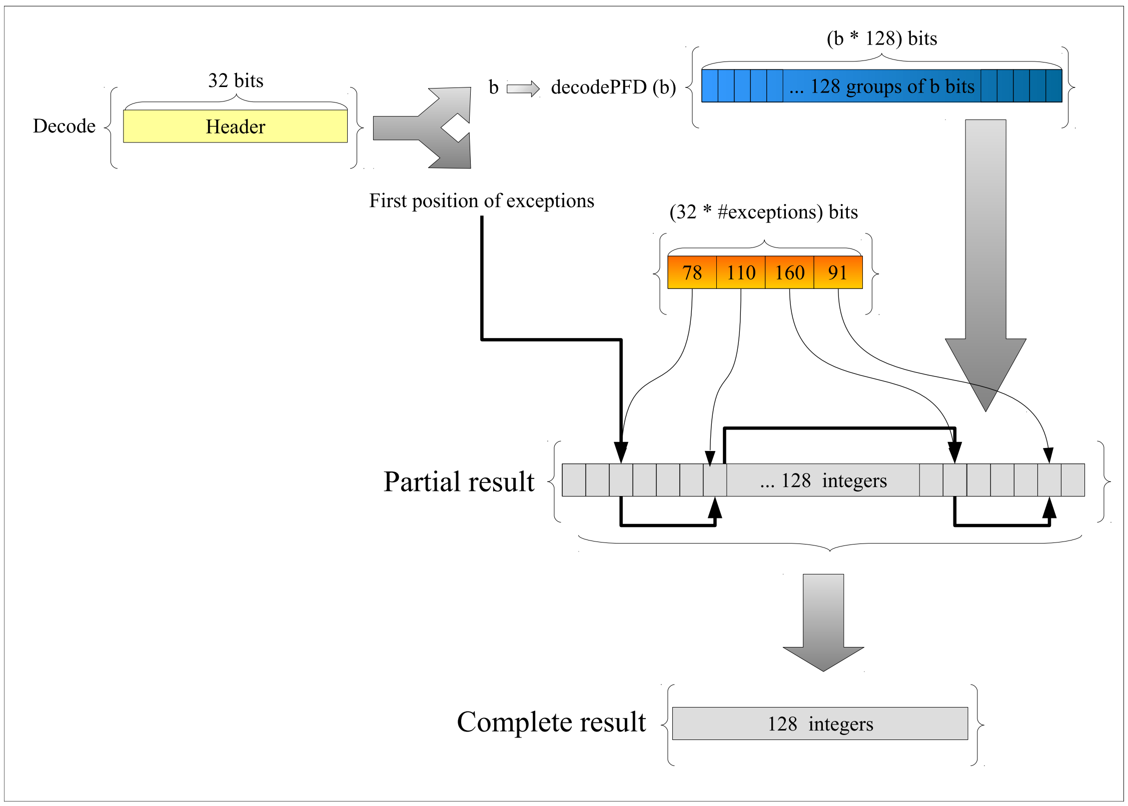


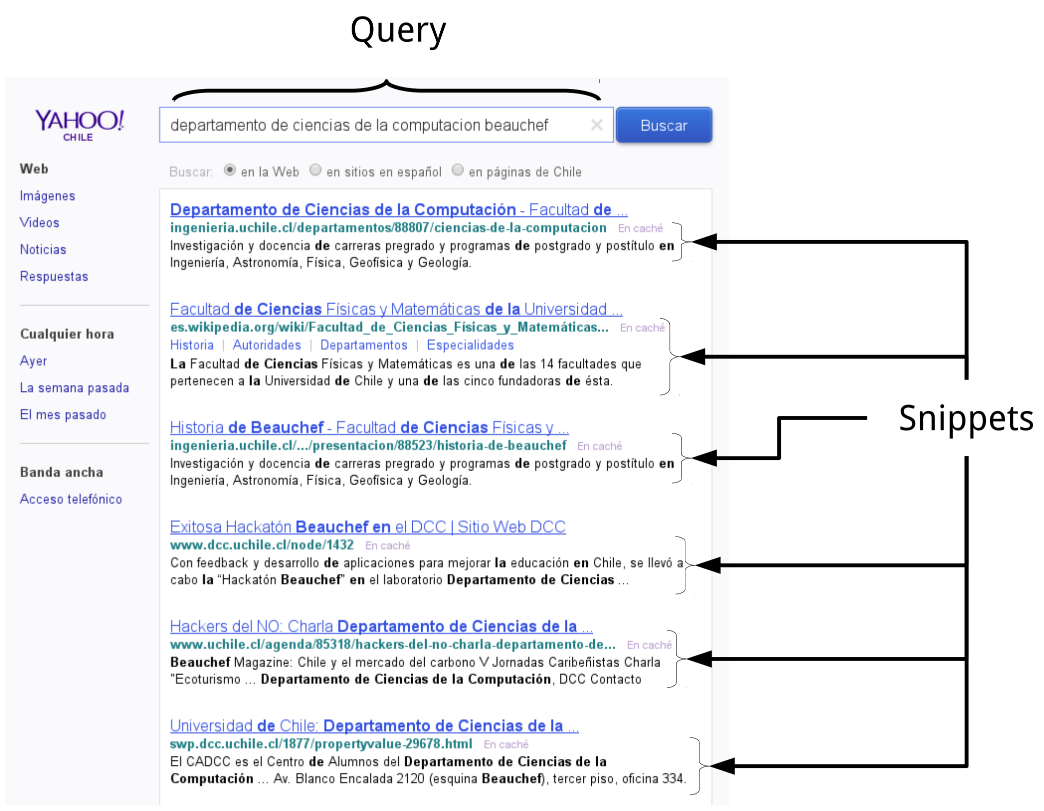
Figure 2.15: Decoding a `PforDelta`block.

```
/*Decode 128 integers of b bits from data and write them in results.*/
void getNumbers(void * data, char b, int * results);

/* Decode the pfordelta block*/
void decodePFD (void * block, int * results){
  int header = ((int *)block)[0];
  char b = (header & 0xfc000000) >> 26 ;

  int exceptionPos = (header & 0x7f) ;
  void * data = (void *)(((int *)(block)) +1);
  int exceptionStart = ((b * 128) + 32) / 32;
  int * exceptions = (int *)(((int *)(block))+(exceptionStart));
  getNumbers(data, b, results);
  int i = 0;
  while( results[exceptionPos] != '0'){
    int offset = results[exceptionPos];
    results[exceptionPos] = exceptions[i];
    exceptionPos += offset;
    i++;
  }
}
```

Figure 2.16: Decoding process for `PforDelta` block.



departamento de ciencias de la computacion beauchef

Departamento de Ciencias de la Computación - Facultad **de** ...
ingenieria.uchile.cl/departamentos/88807/ciencias-de-la-computacion
Investigación y docencia **de** carreras pregrado y programas **de** postgrado y postítulo **en** Ingeniería, Astronomía, Física, Geofísica y Geología.

Facultad **de Ciencias** Físicas y Matemáticas **de la** Universidad ...
es.wikipedia.org/wiki/Facultad_de_Ciencias_Físicas_y_Matemáticas...
Historia | Autoridades | Departamentos | Especialidades
La Facultad **de Ciencias** Físicas y Matemáticas es una **de** las 14 facultades que pertenecen a **la** Universidad **de** Chile y una **de** las cinco fundadoras **de** ésta.

Historia **de Beauchef** - Facultad **de Ciencias** Físicas y ...
ingenieria.uchile.cl/.../presentacion/88523/historia-de-beauchef
Investigación y docencia **de** carreras pregrado y programas **de** postgrado y postítulo **en** Ingeniería, Astronomía, Física, Geofísica y Geología.

Exitosa Hackatón **Beauchef en** el DCC | Sitio Web DCC
www.dcc.uchile.cl/node/1432
Con feedback y desarrollo **de** aplicaciones para mejorar **la** educación **en** Chile, se llevó a cabo **la** "Hackatón **Beauchef**" **en** el laboratorio **Departamento de Ciencias** ...

Hackers del NO: Charla **Departamento de Ciencias de la** ...
www.uchile.cl/agenda/85318/hackers-del-no-charla-departamento-de...
Beauchef Magazine: Chile y el mercado del carbono V Jornadas Caribeñistas Charla "Ecoturismo ... **Departamento de Ciencias de la Computación**, DCC Contacto

Universidad **de** Chile: **Departamento de Ciencias de la** ...
swp.dcc.uchile.cl/1877/propertyvalue-29678.html
El CADCC es el Centro **de** Alumnos del **Departamento de Ciencias de la Computación** ... Av. Blanco Encalada 2120 (esquina **Beauchef**), tercer piso, oficina 334.

Figure 2.17: An example of user-query and the result snippets.

However, to achieve fast extraction, they must use a compression scheme that uses more space than usual compressors. In a more recent work, Ferragina and Manzini [25] study how to store very large text collections in compressed form, such that the documents can be accessed when needed, and show how different compressors behave in such a scenario. One of their main concerns was how compressors can capture redundancies that arise very far apart in very long texts. Their results show that such large texts can often be compressed to just 5% of their original size, which was surprising since usual compression ratio is of about 20%.

2.5.1 Dictionary Compressors with Rank and Select Operations

Dictionary compression consists in storing a dictionary of relevant substrings, and then replacing these strings in a text by the corresponding entry in the dictionary. The most common approaches for dictionary compression update the dictionary entries as compression is carried out, in order to adapt compression to the text at hand. The Lempel-Ziv [52, 53] family of compressors is one of the most important instances of dictionary compressors: many compression tools are based on some kind Lempel-Ziv compression (either algorithm LZ77 [52] or LZ78 [53]). In particular, we review next the compression tools `lzma`, `snappy` and `lz4`, which will be used along this thesis and are based on LZ77 compression.

2.5.2 LZ77 Compression

LZ77 (or sliding window), was the first method devised for Lempel and Ziv [52], which basically tries to detect redundant sequences of symbols in the data. To do this, the method keeps a buffer of fixed size, divided in two sections which are updated as the data is read.

The right part of the buffer is used to read the new symbols, and is known as *lookahead buffer*. The left part of the buffer is called *search buffer* and correspond to the dictionary, it keeps the symbols that have been read previously.

The encoding process is computed as follows: we search for the longest prefix of the *lookahead buffer* that is a substring of the *search buffer*. Then we store a Lempel-Ziv *phrase*(*offset*, *length*, *symbol*) which has 3 components:

- Offset: is the distance between the symbol and its copy in the *search buffer*.
- Prefix Length: length of the copy.
- New symbol : next symbol in the *lookahead buffer*, which makes the corresponding prefix not to be a substring of the *search buffer*.

Then, the look ahead buffer is moved next to the new symbols. If the first symbol of the *lookahead buffer* cannot be found in the *search buffer*, which means that the length of the longest prefix is 0, the Lempel-Ziv *phrase* is (0, 0, symbol), and its called literal.

Example 2.11 The Figure 2.18 shows how the buffer and its two sections are updated while the data is read.

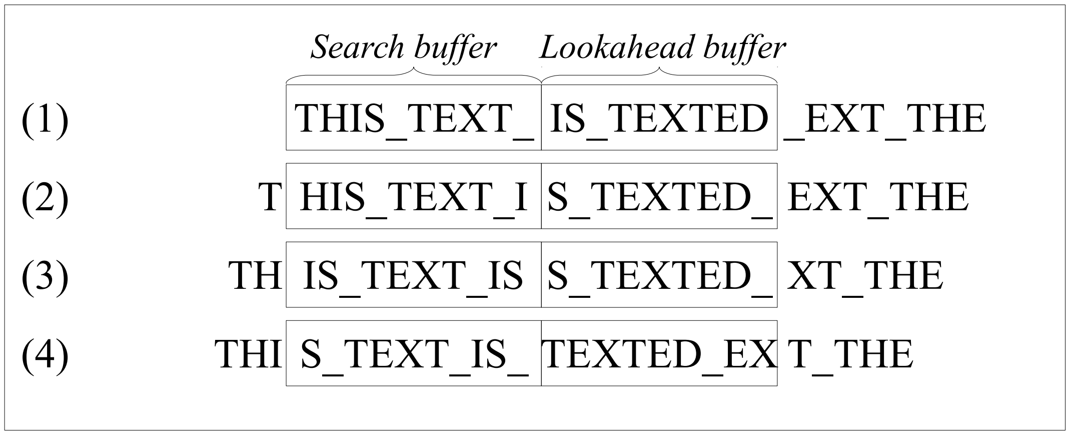

Search buffer
Lookahead buffer
(1) THIS_TEXT_ IS_TEXTED _EXT_THE
(2) T HIS_TEXT_I S_TEXTED_ EXT_THE
(3) TH IS_TEXT_IS S_TEXTED_ XT_THE
(4) THI S_TEXT_IS_ TEXTED_EX T_THE

Figure 2.18: Example of how the buffer of LZ77 is updated.

The decoding process is very simple and efficient: we read sequentially the Lempel-Ziv *phrases* (*offset*, *length*, *symbol*). If the *phrase* is a literal, then we simply write the *symbol* in the output. Otherwise, from the end of the output, we look back *offset* positions and from then copy consecutive *length* symbols at the end of the output. We finally append the *symbol*. In general the size of the *search buffer* is 2^{16} bytes (64 KB), while the *lookahead buffer* is 2^8 bytes (256 bytes). This makes *phrases* be able to be implemented on 32 bits each, and can be therefore efficiently handed.

The following methods, `lzma`, `snappy` and `lz4`, basically share the coding process, which is to find the longest prefix of the *lookahead buffer* in the dictionary (or *search buffer*). The main differences are the dictionary size and the internal structure of the *phrases*.

The base of the decoding process is very similar for all the alternatives mentioned, which is: read the *phrases*, detect if is a copy or a literal, then write it. The minor differences are that some methods encode the *phrases* not aligned to bytes.

2.5.3 `Lzma`

The method `lzma`, which is the principal algorithm of 7-zip[1], is a variant of `LZ77` designed to provide a high compression ratio and a relatively fast decompression speed.

The coding and decoding process are very similar to the `LZ77` process. The main differences are the structure of the *phrases* it generate, and the size of the dictionary. For the *phrases*, `lzma` defines 2 types, instead of 1 as in `LZ77`. If the first bit in the *phrase* is a '0' then the *phrase* is a literal, otherwise a copy. The latter case stores just the *offset* and the *length*. The following is a layout of *phrase* the encoding:

- Literal *phrase*: stores the symbol, coded in range encoding [31].
  - Layout : 0 + code of the symbol.
- Copy *phrase*: stores the pair (*offset*, *length*), using some of the 6 types of representations.

  Type 1: Simple copy, stores (*offset*, *length*), layout : 1 + 0 + *length* + *offset*.

  Type 2: Short copy, where *offset* is the last used and length is 1, layout: 1 +1 +0 +0.

  Type 3: : Long copy (1), uses the last *offset* used, and can store a *length* $\in [2, 273]$, layout: 1+1+0+1+len.

  Type 4: : Long copy (2), uses the second last *offset* used, and can store a *length* $\in [2, 273]$, layout: 1+1+1+0 +len.

  Type 5: : Long copy (3), uses the third last *offset* used, and can store a *length* $\in [2, 273]$, layout: 1+1+1+1+0+len.

  Type 6: : Long copy (4), uses the fourth last *offset* used, and can store a *length* $\in [2, 273]$, layout: 1+1+1+1+1+len.

The encoding of len depends of its value, as can be seen below:

- $len \in [2, 9]$: 0 + 3 bits.
- $len \in [10, 17]$: 1+ 0 + 3 bits.
- $len \in [18, 273]$: 1+ 1 + 8 bits.

[1]http://www.7-zip.org/

A drawback, is that the *phrases* in the output are not aligned to bytes, hence reading them implies several complex bit operations, which increases decompression time.

The dictionary size is large, ranging from 2^{23} (8 MB) to 2^{30} (1,024 MB). This allows the detection of very distant copies, yet the task of searching them in the dictionary increases the time complexity. To achieve a good compression time, `lzma` uses hash tables to quickly find repetitions within the dictionary. Yet due to the large window size, the tables cannot store all the repetitions, producing that some repetitions will not be detected (in other words, not every substring in the window is able to be detected, which can harm compression quality). In spite of this, the large dictionary allows larger chains of symbols to be detected and encoded, achieving higher compression ratios.

2.5.4 Snappy

`Snappy` [1] is another `LZ77` variant, developed by google[2]. It focuses in compression speed and a reasonable compression ratio. The coding and decoding process is the same as described above. The first difference is that instead of using sliding window, `snappy` stores the uncompressed size of the data (max value of $2^{32} - 1$), coded in VByte, at the beginning of the output. Then it stores the compressed data as a stream of *phrases*. This means that `snappy` compresses a large block of information instead of sliding in the data.

The compressed stream uses basically the same scheme that `lzma`. So it stores 2 types of elements: literals and copies. Both elements use theirs first 2 bits to define its type (00 : literal, 01-10-11 : copies). Another difference is that in `snappy` the literals can contain several symbols (from 1 to $2^{32} - 1$) instead of just 1, so after the first 2 bits, a literal stores the amount of symbols that contains (from 1 to $2^{32} - 1$) aligned to bytes, then stores the symbols. There are 2 types of literals:

- Short literals: Store up to 60 symbols, using 6 bits to represent the number of literals (the values 60 -64, are reserved), layout : 0+0+[6 bits] + literals.
- Long literals: Store up to 2^{32} symbols, the values from 60 to 64 of the 6 bits after the initial 00, defines how many bytes are used to represent the amount of literals, layout : 0+0+[6 bits]+ [from 1 to 4 bytes] + literals.

In the case of copies, the *phrases* just store the values *offset* and *length* of the copy, as `lzma`. Depending of the values for *offset* and *length*, there are 3 types of *phrases*:

- Type 1: Use 2 consecutive bytes, using
  - First 2 bits: 01.
  - Next 3 bits: for the *length* of the copy (from 4 to 11).
  - Next 11 bits : for the *offset* of the copy (from 0 to $2^{11} - 1$).
- Type 2: Uses 3 consecutive bytes, using
  - First 2 bits: 10.

[2]http://www.google.com

    - Next 6 bits: for the *length* of the copy (from 1 to 64).
    - Next 16 bits : for the *offset* of the copy (from 0 to $2^{16}-1$).
- Type 3: Uses 5 consecutive bytes, using
    - First 2 bits: 11.
    - Next 6 bits: for the *length* of the copy (from 1 to 64).
    - Next 32 bits : for the *offset* of the copy (from 0 to $2^{32}-1$).

Most of the elements in the data stream are aligned to bytes, which allows for fast reads and writes in the data stream.

The encoding process is slightly different from that of `lzma`, since that the literals can contain several symbols. Also when a copy contains less than 4 symbols, then it is reported as a literal.

2.5.5 `Lz4`

`Lz4` [2], is a method designed to achieve fast decompression speed and a higher compression ratio compared to `snappy`, the difference with the two previous methods is that `lz4` uses just 1 type of *phrase*, as the original `LZ77`. So the coding and the decoding process remain simple.

The layout of a `lz4` *phrase* is as follows:

- Token: uses 1 byte, divided in 2 groups of 4 bits, t_1 and t_2.
    - t_1: Indicates the amount of literals in the *phrase*. If $t_1 = 0$, it means that the *phrase* has no literals, then it reads t_2. If the value is $t_1 = 15$, it means that another bytes are needed to encode the length of the literals. That byte are stored after the token byte, and its information is added to t_1. When the next byte is 255, then another byte is added to the encoding. The process continues as were bytes are needed (it has no limits in the amount of literals that can be stored).
    - t_2: Indicates the value of the length of the copy. If $t_1 = 0$, it means that the *phrase* is not a copy (just a literal). If the value is $t_1 = 15$, it means that another bytes are needed to encode the length of the copy. That byte is stored after the *offset*, and its information is added to t_2. When the next byte is 255, then another byte is added to the encoding. The process continues as were bytes are needed. Notice that `lz4` just stores *offset*≥ 4, then it adds 4 to the result length.
- Length of the literals: it can use from 0 to any amount of bytes needed to store the amount of literals (its used when $t_1 \geq 15$).
- Literals: it can use from 0 to any amount of bytes depending of the amount of literals that are coded in the *phrase*.
- Offset: it can use 2 bytes (if $t_2 = 0$, then this value does not exist), and represent values in $[0, 65, 535]$. It represents the *offset*. Note that 0 is an invalid value,

never used. A 1 means “current position -1 byte”. 65,536 cannot be coded, so the maximum offset value is actually 65,535.

- Length of copy: it can use from 0 to any amount of bytes needed to store the length of the copy (its used when $t_2 \geq 15$).

This scheme provides high adaptability to the context. Also it is aligned to bytes, which means fast access to the data. Additionally, the *phrase* structure allows a very fast decompression when the amount of literals and the copy length are < 15.

2.6 Compressed Text Self-Indexes

Succinct or *compressed* data structures use as little space as possible to support operations as efficiently as possible [37]. Thus, large data sets (like graphs, trees, and text collections) can be manipulated in main memory, avoiding the secondary storage. In particular, we are interested in compressed data structures for text sequences. A *compressed self-index* is a data structure that represents a text in compressed space, supports indexed search capabilities on the text, and is able to obtain any text substring efficiently [37]. They can be seen as compression tools with indexed search capabilities. The following operations have been the building block of many solutions in succinct data structures.

Definition 2.3 Given a sequence $T[1..n]$ over an alphabet $\Sigma = \{1, \ldots, V\}$, we define operation $\mathsf{rank}_c(T, \mathrm{i})$, for $c \in \Sigma$, as the number of occurrences of c in $T[1..\mathrm{i}]$. Operation $\mathsf{select}_c(T, j)$ is defined as the position of the j-th occurrence of c in T.

2.7 Wavelet trees

A *wavelet tree* [27] (`WT` for short) is a succinct data structure that supports rank and select operations, among many virtues [23, 37, 36]

A `WT` representing a text T is a balanced binary search tree where each node v represents a contiguous interval $\Sigma^v = [\mathrm{i}..j]$ of the sorted set Σ. The tree root represents the whole vocabulary. Σ^v is divided at node v into two subsets, such that the left child v_l of v represents $\Sigma^{v_l} = [\mathrm{i}..\frac{\mathrm{i}+j}{2}]$, and the right child v_r represents $\Sigma^{v_r} = [\frac{\mathrm{i}+j}{2}+1..j]$. Each tree leaf represents a single vocabulary term.

Hence, there are V leaves and the tree has height $O(\log V)$. For simplicity, in the following we assume that V is a power of two.

Let T^v be the subsequence of T formed by the symbols in Σ^v. Hence, $T^{root} = T$. Node v stores a bit sequence B^v such that $B^v[l] = \mathtt{1}$ if $T^v[l] \in \Sigma^{v_r}$, and $B^v[l] = \mathtt{0}$ otherwise. Given a `WT` node v of depth i, $B^v[j] = \mathtt{1}$ if the i-th most-significant bit in the encoding of $T^v[j]$ is `1`. In this way, given a term $c \in \Sigma$, the corresponding leaf in the tree can be found by using the binary encoding of c. Every node v stores B^v augmented with a data structure for $\mathsf{rank}/\mathsf{select}$ over bit sequences [37]. The number of bits of the vectors B^v stored at each tree level sum up to n, and including the data structure every level requires $n + o(n)$ bits. Thus, the overall space is $n \log V + o(n \log V)$ bits [27, 37].

Example 2.12 In Figure 2.19 we show an example `WT` for the text "CDEBFEGABBFCHHCDEABG".

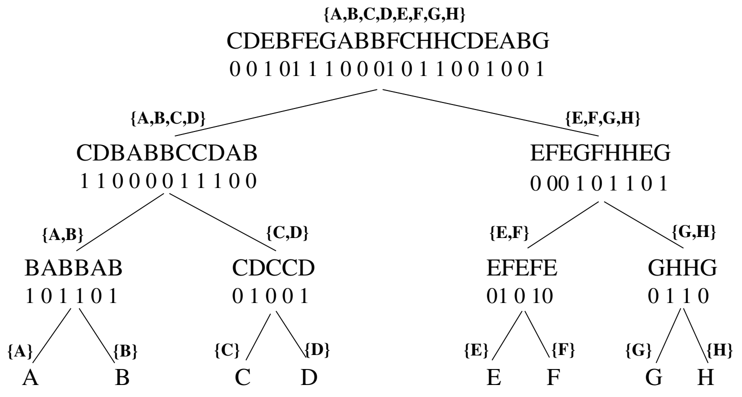


Figure 2.19: Example of a WT.

2.7.1 Supporting Operations

Since a `WT` replaces the text it represents, we must be able to retrieve $T[\mathrm{i}]$, for $1 \leq \mathrm{i} \leq n$. The idea is to navigate the tree from the root to the leaf corresponding to the unknown $T[\mathrm{i}]$. To do so, we start from the root, and check if $B^{root}[\mathrm{i}] = \mathtt{0}$. If so, the leaf of $T[\mathrm{i}]$ is contained in the left subtree v_l of the root. Hence, we move to v_l looking for the symbol at position $\mathsf{rank}_0(B^{root}, \mathrm{i})$. Otherwise, we move to v_r looking for the symbol at position $\mathsf{rank}_1(B^{root}, \mathrm{i})$. This process is repeated recursively, until we reach the leaf of $T[\mathrm{i}]$, and runs in $O(\log V)$ time as we can implement the rank operation on bit vectors in constant time.

To compute $\mathsf{rank}_c(T, \mathrm{i})$, for any $c \in \Sigma$, we proceed mostly as before, using the binary encoding of c to find the corresponding tree leaf. On the other hand, to support $\mathsf{select}_c(T, j)$, for any $c \in \Sigma$, we must navigate the upward path from the leaf corresponding to term c. Both operations can be implemented in $O(\log V)$ time; see [37] for details.

2.7.2 Analysis of Space Usage

The space required by a `WT` is, in practice, about 1.1–1.2 times the space of the text [16]. In our application to IR, this would produce an index larger than the text itself, which is excessive. To achieve compression, we can generate the Huffman codes for the terms in Σ (this is a *word-oriented Huffman coding* [34]) and use these codes to determine the corresponding tree leaf for each term. Hence, the tree is not balanced anymore, but has a Huffman tree shape [16] such that frequent terms will be closer to the tree root than less frequent ones. This achieves a total space of $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits, where $H_0(T) \leq \log V$ is the zero-order empirical entropy of T [30]. In practice, the space is about 0.6 to 0.7 times the text size [16]. However, notice that we have no good worst-case bounds for the operations, as the length of the longest Huffman code assigned to a symbol could be $O(V)$.

2.7.3 Self-Indexes for IR Applications

There have been some recent attempts to apply alternative indexing techniques, such as self-indexes, in large-scale IR systems. In particular, we mention the work by Brisaboa et al. [11] and Arroyuelo et al. [6, 5]. The former [11] concludes that `WT` are competitive when compared with an inverted index for finding all the occurrences of a given query term within a single text. The latter [6, 5] extends [11] by supporting more IR-like operations on a `WT`. Also the work by Brisaboa et al. [10] shows how to use `WT` on Bytecodes [11] for solving ranked document retrieval.

The result is that a `WT` can represent a document collection using $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits while supporting all the functionality of an inverted index. The experimental results in [6] compare with an inverted index storing just `docIDs`, which of course yields a smaller index. However, `WTs` also store extra information, such as the term frequencies and, most important for us here, the compressed text and thus the term-position data.

Recent work [25] also tried to use (among other alternatives) a compressed self-index to compress web-scale texts, in order to allow decompression of arbitrary documents. Their conclusion is that compressed self-indexes still need a lot of progress in order to be competitive with standard compression tools, both in compression ratio and decompression speed. A contribution of this thesis is a compressed self-index that is able to store web-scale texts and is competitive with the best state-of-the-art compressors. We think that this is a step forward in closing the gap between theory and practice in this area [24].

Chapter 3

Indexing for Positional Ranking: Classical Solutions

The ranking process in web search engines is a fundamental step of the search process. This is because it allows the user to find the documents that are more appropriate for their information needs. Although traditionally the tf-idf ranking model is the most used, nowadays search engines use more sophisticated ranking functions. One such example is that of positional ranking functions [13, 48], where the position of the query terms within the resulting documents are used to rank the documents: the closer they are the higher the ranking. The rationale behind positional ranking is that documents, where the query terms occur close to each other might be ranked higher, as this could indicate that the document is highly relevant to the query.

Another important aid used by nowadays web search engines is that of result snippets: a relevant portion of the document, which is used to help the user to decide about the relevance of a result.

In this chapter we review the state of the art for position indexing and snippet generation, and show experimental results that will be used to determine the improvements introduced by our contributions.

3.1 Basic Query Processing Steps for Positional Ranking and Snippet Extraction

From now on we assume a search engine where positional ranking is used to score documents, and where snippets must be generated for the top-scoring documents. Thus, solving a query involves the following steps:

1. **Query Processing Step:** Given a user query, use an inverted index to obtain the top-k_1 documents according to some standard query processing approach (e.g.,

DAAT) and ranking function (e.g., BM25).

2. **Positional Ranking Step:** Given the top-k_1 documents obtained on the previous step, obtain the positions of the query terms within these documents. Then re-rank the results using a positional ranking function [13, 49].
3. **Snippet Generation Step:** After the re-ranking of previous step, obtain snippets of length s for the top-k_2 documents, for a given $k_2 \leq k_1$.

An illustration of this process is shown in Figure 3.1, where term positions are obtained from a positional index (to be described next in this chapter) and text snippets are obtained from the compressed text database.

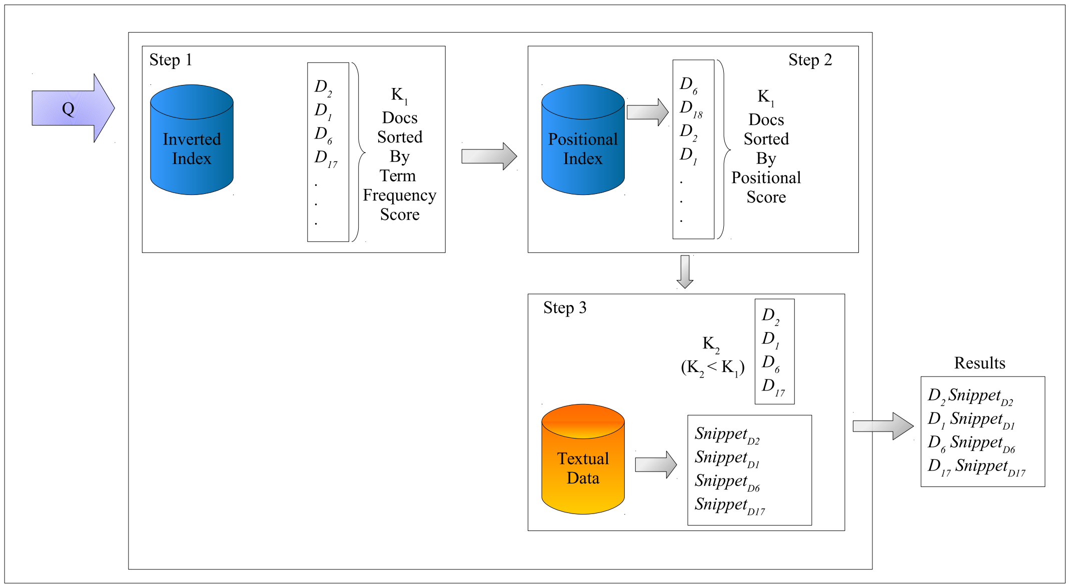

Q
Step 1
Inverted Index
D_2 D_1 D_6 D_{17} . . .
K_1 Docs Sorted By Term Frequency Score
Step 2
Positional Index
D_6 D_{18} D_2 D_1 . . .
K_1 Docs Sorted By Positional Score
Step 3
K_2 ($K_2 < K_1$)
D_2 D_1 D_6 D_{17}
Textual Data
$Snippet_{D2}$ $Snippet_{D1}$ $Snippet_{D6}$ $Snippet_{D17}$
Results
D_2 $Snippet_{D2}$
D_1 $Snippet_{D1}$
D_6 $Snippet_{D6}$
D_{17} $Snippet_{D17}$

Figure 3.1: Illustration of the query process with positional ranking and snippet generation.

For instance, $k_1 = 200$ (as in [49]) and $k_2 = 10$ (as in most commercial search engines) are typical values for the query parameters. We assume $s = 10$ in this thesis. The different values for these parameters should be chosen according to the trade-off between query time and search effectiveness that we want to achieve. Step 2 is usually supported by a positional inverted index [29, 13, 49]. Step 3 is supported by compressing the document collection and supporting the extraction of arbitrary documents. We will describe these processes in detail in this chapter. The focus of this thesis is on alternative ways to implement the last two steps. Next, we evaluate experimentally the current solutions on the state of the art of positional indexing.

3.2 Experimental Setup and Dataset Description

Along this thesis, we use the following experimental setup. The machine where we ran our experiments is an HP ProLiant DL380 G7 (589152-001) server, with a Quadcore Intel(R) Xeon(R) CPU E5620 @ 2.40GHz processor, with 128KB of L1 cache, 1MB of L2 cache, 2MB of L3 cache, and 96GB of RAM, running version 2.6.34.8-68.fc13.i686.PAE of Linux kernel.

We use the TREC GOV2 document collection, with about 25.2 million documents and about 32.86 million terms in the vocabulary. We work just with the text content of the collection (that is, we ignore the html code from the documents). This requires about $130,048$MB in ASCII format. When we represent the terms as integers, the resulting text uses 91,634 MB. We use a subset of 10,000 random queries from the TREC 2006 query log. All methods were implemented using C++, and compiled with `g++ 4.4.5`, with the full optimization flag `-O5`.

To compare the different schemes proposed in this thesis, we use two criteria. The first one is the compression ratio, which is the size of the compressed structure divided by the size of the uncompressed structure multiplied by 100. The second one is the time per query, which is the average time to answer a query with the defined scheme.

3.3 The Baseline: Positional Inverted Lists and Compressed Textual Data

This section describes and evaluates the baseline approaches to support term-position indexing and snippet extraction. These will be the starting points for the proposals of this thesis.

3.3.1 Supporting the Query-Processing Step

As we have said, query processing is supported by inverted indexes. To achieve competitive space and time with an inverted index, inverted lists are usually divided into chunks (of usually 128 `docID`s each) and implemented using layers. The first layer stores the `docID`s and the second one stores frequencies. See Figure 3.2 for an illustration.

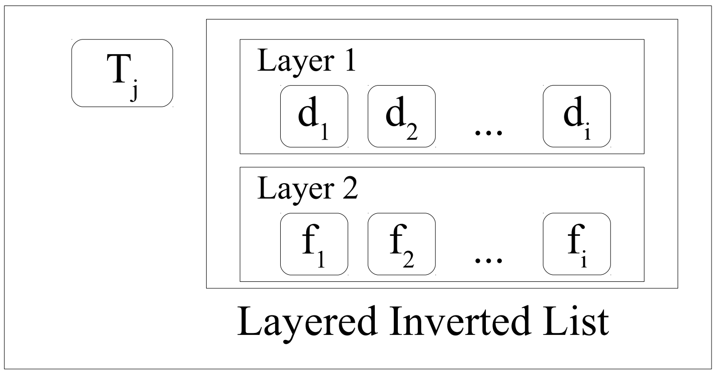


Figure 3.2: Diagram of the layer implementation.

An advantage of this layered scheme is that every layer can be compressed differently, using the best alternative for each layer. In our experimental setting, these alternatives are `PforDelta` (Section 2.4.3) for the `docID` layer and S16 (Section 2.4.2) for the frequency layer, just as in [49]. Another advantage is that data is decompressed just when necessary at search time. A chunk in the second layer is decompressed only when a given `docID` in the first layer is a candidate to be in the top-k_1. This process results in a fairly competitive query time to obtain the top-k_1 documents.

In our experiments the index for `docIDs` and frequencies for the GOV2 collection requires $47,854$ MB of space in uncompressed form. Using `PforDelta` compression for `docIDs` and S16 for frequencies, the space usage is reduced to $9,739$ MB, achieving a compression ratio of about 20%. Notice also that the index uses 0.11 times the space of the input text.

For query processing, we assume the well-known Document-at-a-Time (DAAT) approach, with the BM25 ranking [13] to obtain the top-k_1 most relevant documents in Step 1. BM25 is one of the most used and effective term-frequency scoring function. Given a query Q with terms $q_1, ..., q_n$ and a document $\mathcal{D}$, the score is computed as follows:

$$score_{BM25}(\mathcal{D}, Q) = \sum_{\mathrm{i}=1}^{n} IDF(q_\mathrm{i}) \cdot \frac{f(q_\mathrm{i}, \mathcal{D}) \cdot (a+1)}{f(q_\mathrm{i}, \mathcal{D}) + a \cdot (1 - b + b \cdot \frac{|\mathcal{D}|}{avg\mathrm{dl}})},$$

where $f(q_\mathrm{i}, D)$ is the frequency of term q_i in document $\mathcal{D}$, $|\mathcal{D}|$ is the number of words of document $\mathcal{D}$, and $avg\mathrm{d}l$ is the average document length in the text collection. Parameters a and b are for advanced optimization, typically $a \in [1.2, 2.0]$ and $b = 0.75$. Finally, $\mathrm{IDF}(q_\mathrm{i})$ is the IDF (inverse document frequency) weight of the query term q_i. It is usually computed as:

$$\mathrm{IDF}(q_\mathrm{i}) = \log \frac{N - n(q_\mathrm{i}) + 0.5}{n(q_\mathrm{i}) + 0.5},$$

where N is the number of documents in the collection, and $n(q_\mathrm{i})$ is the number of documents that contain term q_i.

In Table 3.1 we show the average query time (in milliseconds per query) for the initial query processing Step 1, obtained from our experiments. We show results for two types of queries: traditional AND queries (using DAAT query processing) and the BMW OR approach from [19], which is one of the most efficient current solutions for disjunctive (OR) queries.

Notice that the query time for AND is almost constant (within two decimal digits) with respect to k_1. The process to obtain the top-k_1 documents uses a heap (of size k_1). However, operating the heap takes negligible time, compared to the decompression of `docIDs` and the DAAT process. BMW OR, on the other hand, is an early-termination technique, and thus k_1 impacts the query time.

Table 3.1: Experimental results for the initial query processing step (Step 1) for AND and OR queries. In both cases, BM25 ranking is used.

top-k_1	DAAT AND (ms/q)	BMW OR [19] (ms/q)
50	14.75	35.70
100	14.77	43.39
150	14.80	47.90
200	14.81	51.74
300	14.81	58.19

3.3.2 Supporting the Positional-Ranking Step

Positional inverted lists (`PIL`s, for short) are the standard approach for indexing in-document position data in search engines [29, 13, 8].

`PIL`s add another layer to the layered structure described in Section 3.3.1. See Figure 3.3 for a reference. As for `docID`s and frequencies, we divide the positions in chunks, which now have variable size: if a `docID` chunk stores n `docID`s, the corresponding positional chunk stores as many positions as the sum of all the frequencies stored in the frequency chunk. Hence, because of the variable size of the positional chunks, each chunk of the frequency layer stores a pointer to the corresponding positional chunk.

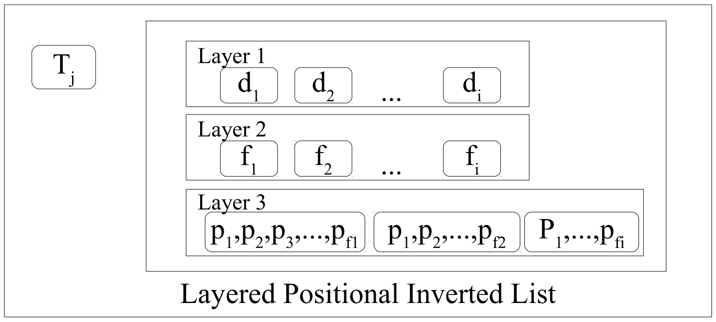

T_j
Layer 1
d_1 d_2 ... d_i
Layer 2
f_1 f_2 ... f_i
Layer 3
$p_1,p_2,p_3,\dots,p_{f1}$ $p_1,p_2,\dots,p_{f2}$ $P_1,\dots,p_{fi}$
Layered Positional Inverted List

Figure 3.3: A single block in the layered implementation of inverted lists, storing `docID`s, frequencies and positions.

To obtain positional data at query time, after we obtain the top-k_1 `docID`s for a given query, we identify the positional chunks containing the desired positional index entries. Then, these positional chunks are *fully decompressed*, and the corresponding positions are obtained. A drawback here is that we need to decompress the entire positional chunk, even if we only need a single entry in it. Thus, we might end up decompressing, in the worst case, k_1 positional chunks in each of the inverted lists involved in the query.

After obtaining the positions of all query terms within the top-k_1, we proceed to re-rank them using a positional ranking score. Particular positional ranking functions are out of the scope of this thesis: any function than uses term positions could be used. In our experiments, we use the scoring model proposed by S. Büttcher and C. Clarke [14, 40], which is defined as follows. For each document we fetch the positions of all the query terms within the top-k_1 documents. Each term is associated an accumulator that contains the term proximity score, which is computed as follows: for every pair of consecutive terms $T_{\rm i}$, T_j ($T_{\rm i} \neq T_j$) in the query Q, we obtain all the positions of those terms ($P_{T{\rm i}}$,P_{Tj}) in the document $\mathcal{D}$ and modify the accumulators for $T_{\rm i}$ and T_j as:

$$acc(T_{\rm i}) \leftarrow acc(T_{\rm i}) + w_{Tj} \cdot ({\rm di}st(P_{T{\rm i}} + P_{Tj}))^{-2}$$
$$acc(T_j) \leftarrow acc(T_j) + w_{T{\rm i}} \cdot ({\rm di}st(P_{T{\rm i}} + P_{Tj}))^{-2}$$

where w_{T_x} is the IDF score of T_x (the same as in BM25). After computing all the accumulators, the score of the document $\mathcal{D}$ is:

$$score_{BM25+Pos}(\mathcal{D}) = score_{BM25} + \sum_{T \in Q} min(1, w_t) \cdot \frac{acc(T) \cdot (a+1)}{acc(T) + b},$$

where a and b are the same as in the BM25 formula.

It is important to remark that the space needed for in-document positions is usually high, since we must store the positions of all the occurrences of a term. For instance, in our experimental setting the total number of positions to be stored is 23,991,731,648. This is about twice the number of `docID`s and frequencies that need to be stored in the inverted lists (about 12,512,013,184 integers). In uncompressed state, this means about 91,521 MB for positions, which is about twice the space used by the uncompressed `docID`s and frequencies. Moreover, position data usually cannot be compressed as easily as in the case of `docID` and frequencies, which makes the situation even worse.

As we already said, the access pattern for position data is much sparser than that for `docID`s and frequencies, since positions must be obtained only for the top-k_1 documents. Thus, just a few positions are decompressed from the `PIL` in each query. Given this sparse access pattern and the high space requirement of positions (as discussed above), it is better to use compression methods with a good compression ratio, like Golomb/Rice compression. These are slower to decompress, yet the fact that only a few positions are decompressed should not impact considerably in the overall query time. According to [49] and further personal communications with the main author of that work, the most effective compression techniques are Rice and S16. In Table 3.2 we show experimental results for obtaining positions with the baseline `PIL`s, using the two compression schemes selected, Rice and S16. We also show query times for different values of k_1, namely 50, 200 and 300 (the experiments in [49] only use $k_1 = 200$).

As we can see, Rice requires only about 90% the space of S16, but takes twice as much time. Comparing the query times of Step 2 for Rice and S16 with the query times of Step 1 in Table 3.1, we can see that position extraction is a small fraction of the overall time. Hence, we can use Rice to compress `PIL`s and obtain a better space usage with only a minor increase in query time. For Rice, `PIL`s use 2.91 times the space of the inverted index that stores `docID`s and frequencies. For S16, this number is 3.22.

Table 3.2: Experimental results for extracting term-position data (Step 2).The compression ratio is computed according to the size of the uncompressed text, which is 91,634 MB.

Approach	Compression Scheme	Space (MB)	Compression Ratio	Position extraction time (msec/query)		
				$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
`PIL`s (no text)	S16	31,338	34.19	0.74	1.75	2.51
	RICE	28,373	30.96	1.28	3.27	5.57

3.3.3 Supporting the Snippet Generation Step

In order to support snippet generation, web search engines must store a copy of the entire web in their servers. This requires, of course, considerable space. To compress the text collection and support decompressing arbitrary documents, a simple alternative that is used by several state-of-the-art search engines — for instance Lucene [17] — is to divide the whole collection into smaller text blocks, which are then compressed separately. The block size offers a time-space trade-off: larger blocks yield better compression, although decompression time is increased. Given the popularity [17, 25] and simplicity of this approach, we use it as the baseline for the compressed text.

Table 3.3 shows experimental results for the baseline for compressed textual data. Just as in [25], we divide the text into blocks of 0.2MB, 0.5MB or 1.0MB, and compress each block using different standard text compression tools. In particular, we show results for `lzma` (which gives very good results in [25], so it serves as a comparison with the results in that work) and Google's `snappy` compressor [1], which is an LZ77 compressor that is optimized for speed rather than compression ratio. We also show results for `lz4` [2], which is another variant of LZ77 compression improved for speed. These three compressors offer the most interesting trade-offs among the alternatives we tried.

As it can be seen, `lzma` achieves much better compression ratios than `snappy` and `lz4`. Also, in all cases the compression ratio improves as we increase the block size. This is because more text regularities can be detected. If we now consider the text as a whole (i.e., without the block structure, which is not useful for snippet generation, but for archival purposes) the compressed space achieved for the whole text is 8,133 MB for `lzma`, 27,388 MB for `snappy` and 29,808 MB for `lz4`. Notice that the compression ratio for `lzma` is similar to that reported in [25]. Even when `snappy` uses less space than `lz4` compressing the whole text, separating the web in blocks shows that `lz4` has better performance in space and time. The differences in extraction time are also considerable, with `lz4` being faster than the other two alternatives, especially against `lzma`. Note that [25] reports a decompression speed of about 35MB/sec for `lzma`. However, to obtain a given document we must first decompress the entire block that contains it. Hence, most of the 35MB per second do not correspond to any useful data. In other words, this does not measure effective decompression speed for our scenario, and thus

Table 3.3: Experimental results for the snippet extraction phase (Step 3).

Compressor	Block size	Space usage	Compression Ratio	Average snippet extraction time (ms/q) for different values of k_2		
	(MB)	(MB)		$k_2 = 10$	$k_2 = 30$	$k_2 = 50$
`lzma`	0.2	14,987	16.35	29	84	136
	0.5	13,489	14.72	63	181	292
	1.0	12,682	13.84	117	335	540
`snappy`	0.2	34,576	37.73	2	6	9
	0.5	34,426	37.57	5	14	23
	1.0	34,390	37.53	10	28	46
`lz4`	0.2	30,405	33,18	1	4	7
	0.5	30,070	32,81	3	10	16
	1.0	29,953	32,68	7	19	31

we report per-query times rather than MB/s for both methods.

Finally, notice that when we use `lzma` for the text, the space usage for the whole collection is much smaller than the space of positions. This is because compressors can take advantage of the text regularities, wherever they are. Positions, on the other hand, are stored separately for each term, so inter-term text regularities cannot be detected and compressed. A relevant question here is: How can one represent terms positions, in such a way that these text regularities are conserved? We answer this question as one of the most important contributions of this thesis.

3.4 Proposed Solution: Computing Term Positions from Textual Data

The main conclusion from the previous section is that position and text data have high space requirements. The size of the two structures combined is 7 times the space used by the `docID` and frequency inverted index, which becomes a bottleneck [49].

This thesis focuses on alternative approaches to perform the aforementioned two-step document ranking process and the query snippet-generation phase, which are Step 2 and Step 3 in Section 3.1. The aim is to optimize both space and query processing time. One important feature of position data is that it only needs to be accessed for a limited number of promising documents, say a few dozens or hundreds of documents. This access pattern differs from that for document identifiers and term frequencies, which are accessed more frequently, making access speed much more important. For position data, on the other hand, we could consider somewhat slower but smarter alternative representations without losing too much efficiency at query time [49].

In this thesis, we push this idea further and consider not storing the position data (i.e, the positional index) at all. Instead, we propose to compute positions *on the fly* from a compressed representation of the text collection. We will study two alternative approaches to compressing the text collection:

1. *Wavelet trees* [27], which are succinct data structures from the combinatorial pattern matching community.
2. Compressed document representations, that support fast extraction of arbitrary documents.

It has been shown that, compared to positional indexes, web-scale texts can often be compressed in much less space [25]. More importantly, in our proposal the compressed text can be used for both positional re-ranking and snippet generation, saving additional space.

An example of the procedure that we propose is shown in Figure 3.4.

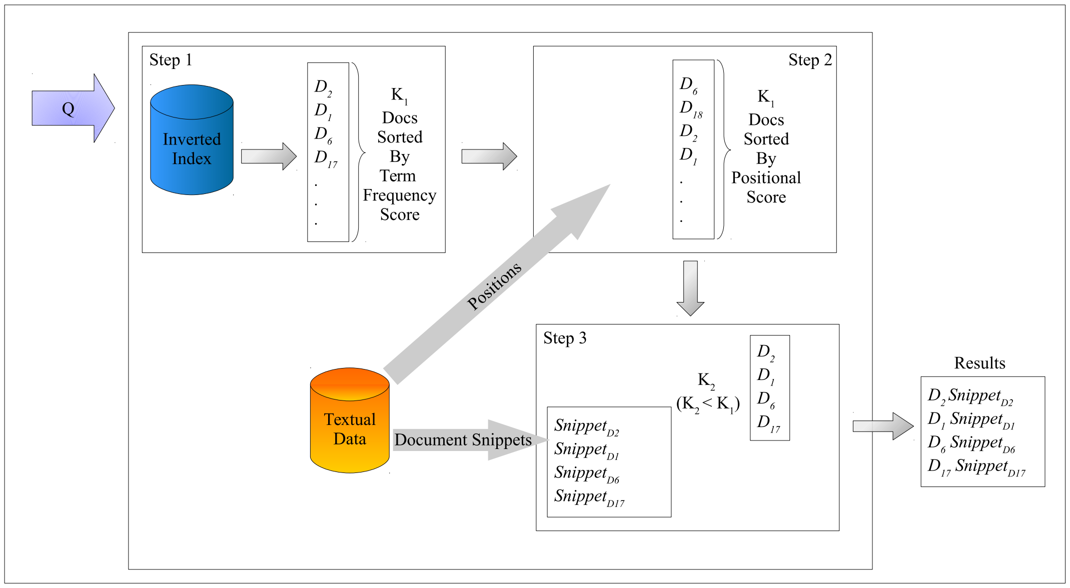

Q
Step 1
Inverted Index
D_2 D_1 D_6 D_{17} . . .
K_1 Docs Sorted By Term Frequency Score
Step 2
D_6 D_{18} D_2 D_1 . . .
K_1 Docs Sorted By Positional Score
Positions
Step 3
K_2 ($K_2 < K_1$)
D_2 D_1 D_6 D_{17}
Textual Data
Document Snippets
$Snippet_{D2}$ $Snippet_{D1}$ $Snippet_{D6}$ $Snippet_{D17}$
Results
D_2 $Snippet_{D2}$
D_1 $Snippet_{D1}$
D_6 $Snippet_{D6}$
D_{17} $Snippet_{D17}$

Figure 3.4: Illustration of the query process from Section 3.1, using the textual data to obtain positions and for the snippet generation.

Thus, solving a query with our proposal, includes the following steps:

1. **Query Processing Step:** Given a user query, use an inverted index to obtain the top-k_1 documents according to some standard query processing approach (e.g., DAAT) and ranking function (e.g., BM25).
2. **Positional Ranking Step:** Given the top-k_1 documents obtained on the previous step, obtain the in-document positions for the query terms from the documents themselves. Then re-rank the results using a positional ranking function [13, 49].
3. **Snippet Generation Step:** After the re-ranking of previous step, obtain snippets of length s for the top-k_2 documents, for a given $k_2 \leq k_1$.

One concern is how these alternatives impact query processing speed, as we must decompress documents and then search for the query terms within them. We will study the resulting trade-offs between running time and space requirement.

Thus, *to index or not to index position data*, that is the research question that we hope to answer in this thesis. To our knowledge, such alternative approaches for storing positional data have not been rigorously compared before. Our main result is that we can store all the information needed for query processing (i.e., document identifiers, term frequencies, position data, and text) using space close to that of state-of-the-art positional indexes (which only store position data and thus cannot be used for snippet creation), with only a minor increase in query processing time. Thus, we provide new alternatives for practical compression of position and text data, outperforming the recent approaches from [42].

Chapter 4

A *Wavelet Tree* for Computing Positions

In this chapter we explore the alternative of representing the text collection using a *Wavelet Tree* (`WT`) data structure. We then use the `WT` functionality to obtain both, term-positions and snippets. This data structure replaces the positional inverted lists and the text collection, aiming at a better space usage.

Let $T = \$D_1\$D_2\$\cdots\$D_N\$$ be the text obtained from the concatenation (in arbitrary order) of the documents in the collection, where the documents are separated by a special symbol '\$'. We represent a text T with a `WT` to obtain term positions and text snippets. Given a position i in T, one can easily obtain both the `docID` of the document that contains $T[\mathrm{i}]$ as $rank_{\$}(T, \mathrm{i})$, and the starting position of a given document j by means of $select_{\$}(T, j)$ [6].

4.1 Byte-Oriented Huffman `WT`

Instead of a bit-oriented `WT` (as the one explained in Section 2.6), we use the byte-oriented representation from [11], using a Huffman encoding of the words, which is the most efficient alternative reported in there. The idea is to first assign a Huffman code to each vocabulary term [34]. Then, we store the most significant *byte* of the encoding of each term in array B^{root}. That is, each `WT` node v stores an array of bytes B^v, instead of bit arrays as in Section 2.6. Next, each term in the text is assigned to one of the children of the root, depending on the first byte in the encodings. Notice that in this way the `WT` is 256-ary. See [11] for details.

To support rank and select, we use the simple approach from [11]. Given a `WT` node v, we divide the corresponding byte sequence B^v into super-blocks of s_b bytes each. For each super-block we store 256 *super-block counters*, one for each possible byte. These counters tell us how many occurrences of a given byte there are in the text up to the last position of the previous super-block. Also, each super-block is divided into blocks of b bytes each. Every such block also stores 256 *block counters*, similarly as before. The difference is that the values of these counters are local to the super-block, hence less bits are used for them. To compute $\mathsf{rank}_c(T, \mathrm{i})$, we first compute the super-block j that contains i, and use the super-block counter for c to count how many c there are in T up to super-block $j-1$. Then we compute the block i' that contains i and add (to the previous value) the block counter for c. Finally, we must count the number of c within block i'. This is done with a sequential scan over block i'. This block/super-block structure allows for time-space trade-offs. In our experiments we use $s_b = 2^{16}$. Hence, super-block counters can be stored in 16 bits each. We consider $b = 1$ KB, $b = 3$ KB and $b = 7$ KB. Operation select is implemented by binary searching the super-block/block counters; thus no extra information is stored for this [11].

4.2 Obtaining Term Positions from a WT

To obtain position data assume that, given docID i for a top-k_1 document and a query term t, we want to obtain the positions of t within D_i. A simple solution could be to extract document D_i from the WT, and then search for t within it. However, a more efficient way is to use operation select to find every occurrence of t within D_i, hence working in time proportional to the number of occurrences of the term (and not the document length). Let d be the starting position for document D_i in T. Hence, there are $r \leftarrow \mathsf{rank}_t(T, \mathrm{d})$ occurrences of t before document D_i, and the first occurrence of t within D_i is at position $j \leftarrow \mathsf{select}_t(T, r+1)$, the second occurrence at position $j' \leftarrow \mathsf{select}_t(T, r+2)$, and so on. Overall, if o is the number of occurrences of t within D_i, then we need 1 rank and $o+1$ selects to find them. An illustration of this process can be seen in Figure 4.1.

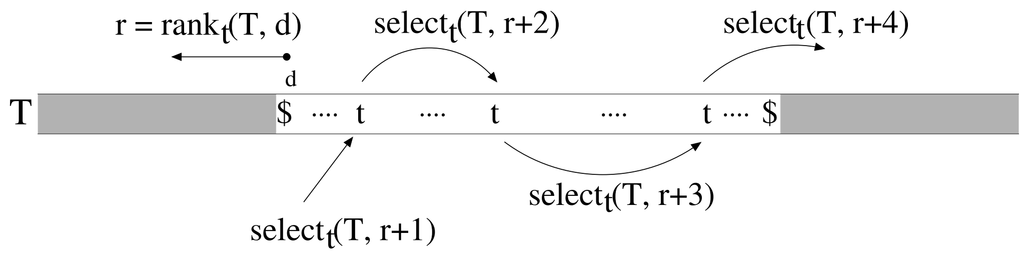


Figure 4.1: Illustration of the position-extraction process in a WT , using operation select.

4.3 Experimental Results

We fully implemented the byte-oriented WT and the position extraction process. The aim was an implementation able to deal with large texts. Indeed, the text indexed in our experiments is the largest text indexed with a succinct/compressed data structure we are aware of in the literature.

In Table 4.1 we show the experimental trade-offs obtained for `WT`, for the different block sizes tested (see the rows "`WT`(7 KB)" and "`WT`(1 KB)"). For the sake of comparison, we also show the results for `PIL`s from the previous chapter.

Note that `WT` (1 KB) obtains better times than `PIL`s, yet requiring considerably more space. An advantage of the `WT` structure is that it can search for positions and get the snippets of the documents within the same space. `WT` (7 KB), on the other hand, is slower than `PIL` (Rice) and uses more space. Even though the `WT`, includes the textual data, its high space usage could leave it out of consideration on schemes where the text is not necessary. Next, we introduce extra improvements to make them competitive.

Table 4.1: Experimental results for extracting term-position data using a `WT` . The compression ratio is computed according to the size of the uncompressed text, which is 91,634 MB.

Approach	Compression Scheme	Space (MB)	Compression Ratio	Position extraction time (msec/query)		
				$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
`PIL`s (no text)	S16	31,338	34.19	0.74	1.75	2.51
	RICE	28,373	30.96	1.28	3.27	5.57
Compressed self-indexes	`WT`(7 KB)	40,534	44.23	1.94	6.85	9.80
	`WT`(1 KB)	56,917	62.11	0.33	1.15	1.75
	`WT`(7 KB) + `lzma`	19,628	21.42	19.25	68.36	97.71
	`WT`(1 KB) + `lzma`	42,359	46.23	7.22	24.97	35.57
	`WT`(7 KB) + `snappy`	25,122	27.42	14.35	51.02	74.56
	`WT`(1 KB) + `snappy`	46,778	51.05	2.07	7.32	10.47
	`WT`(7 KB) + `lz4`	24,911	27.18	14.55	51.05	74.60
	`WT`(1 KB) + `lz4`	46,600	50.85	2.08	7.31	10.48

4.4 Achieving Higher-Order Compression with the `WT`

Basically, `WT`s are zero-order compressors, which explains their high space usage. To achieve higher-order compression, notice that B^{root} contains the most significant byte of the Huffman encodings of the original terms in the text. Thus, the original text structure is at least partially preserved in the structure of B^{root}, which might thus be as compressible as the original text. A similar behavior can be observed in internal nodes for the remaining bytes that form the encodings of the terms. Thus, we propose to compress the blocks of B^v in each `WT` node v by using standard compressors, based on LZ77 compressors (because decompression time is crucial for time efficiency in our case). Table 4.1 shows results for `lzma`, `snappy` and `lz4`, the best compressors we tried.

Notice that `WT` (7 KB) + `lzma` achieves 19,628 MB, almost half the space used by `WT` (7 KB). The time to obtain positions becomes, on the other hand, an order of magnitude slower. `WT` (7 KB) + `snappy` achieves slightly better times, using space that is smaller to that of `PIL`s.

Overall, although the times are slower than that of `PIL`s, this significant reduction in space could make `WT`s competitive in scenarios where storing the text is relevant.

Chapter 5

Computing Term Positions from the Compressed Text

In this chapter we explore the alternative of obtaining the positional information directly from the compressed text. We test several compression schemes, aiming at a better space usage.

5.1 Using Standard Compressors

Using standard text compressors, our next approach is to obtain positions using the baseline for generating snippets from Section 3.3.3. At search time, the top-k_1 documents are obtained from their corresponding blocks. Hence, in the worst case we must decompress k_1 text blocks, compared with the (worst-case) k_1 positional chunks for each query term from the `PIL`s. Then, the query terms are sought within these documents, obtaining their positions. Since we are looking for single terms, no sophisticated text search algorithm is used (like KMP, for instance). We just carry out a single scan on the document, and for each text position we check whether it is on the query terms or not. For the snippet extraction step, no further decompression is needed, because the documents for the top-k_2 are already decompressed.

In Table 5.1 (on page 42) we show experimental results for this approach, using `lzma`, `snappy` and `lz4` compressors, and blocks of size 200 KB. We also compare with the times and space obtained using `PIL`s (from Chapter 3) and the best alternatives for `WT`s (from Chapter 4). We can see that using `lzma`, we can store positions and text in about half the space of `PIL` (the latter just storing positions). Although this is promising, this approach is two orders of magnitude slower than the positional index, which limits its use in real systems. Actually, the time is about 10 times slower than that of Step 1 (recall Table 3.1 on page 28). If we use `snappy` instead, we obtain an index that is 21.86% larger than `PIL` (Rice), or even using the results of `lz4` which is 7.16% larger than `PIL` (Rice). The times to obtain positions are about 6 times slower than using `PIL`s, (which might be still acceptable in some cases since it corresponds to about 0.5 times the time of Step 1 in the query process).

Even though the space usage achieved with `lzma` is competitive, the time needed to obtain the positions is too high. Hence the solution in not practical for web search engines. In what follows, we shall try several approaches to achieve (as much as possible) the space usage of `lzma`, yet with a practical time for obtaining positions.

5.2 Using Zero-Order Compressors with Fast Text Extraction

An alternative to compressing the text that could support faster position lookups is the approach from Turpin et al. [45]. The idea is to first sort the vocabulary according to the term frequencies, and then assign term identifiers according to this order. In this way, the term identifier `0` corresponds to the most-frequent term in the collection, `1` to the second-most-frequent term, and so on. The document collection is then represented as a single sequence of identifiers, where each term identifier is encoded using VByte [3]. Note that the 128 most frequent terms in the collection are thus encoded in a single byte. Actually, the work in [45] uses a move-to-front strategy to store the encodings: the first time a term appears in a document, it is encoded with the original code assigned as before; the remaining appearances are represented as an offset to the previous occurrence of the term. We also use this approach in our experiments.

By using either an integer compression scheme, such as VByte or VNibble (a variant of VByte that represents any integer with a variable number of nibbles, i.e., half bytes, recall Section 2.4.1) for the text, or a word-based compression scheme like byte-oriented word Huffman [34], we are able to decompress *any text portion* very efficiently. No text blocks are needed this time, but just a small table indicating the starting position of each document. Table 5.1 shows the resulting trade-offs for the alternatives described until now, compared with the results obtained using `PIL`s (from the previous chapter). We also show results for VNibble and byte-oriented Huffman.

Notice that we improve the position extraction time significantly, making it competitive with `PIL`s. This shows that being able to extract just the desired k_1 documents is an important fact (since we save time that is otherwise wasted when decompressing a whole block with standard compressors). The higher space usage, however, is a concern. Yet, note that we also represent the text within this space, not just the position data as in `PIL`s. We also tried other compression schemes, such as `PforDelta` and S9, obtaining poorer compression ratios and similar decompression speed.

In our experimental results (see Table 5.1, rows "VByte", "VNibble" and "Byte-oriented Huffman"), we obtain space savings of about 10% for VNibble, and about 2% for byte-oriented Huffman, in both cases compared to VByte's performance. Also, notice that we are now able to use space close to that of `snappy` (with blocks of 200 KB), yet with a better query time.

The faster position extraction time obtained is due to two facts. First, byte-oriented Huffman has a fairly fast decompression speed [34], and methods like VByte and VNibble are able to decompress hundred of million integers (which in our case correspond to terms) per second [50]. Second, these methods are able to decompress just the desired documents (as we already said), without negative impact on compressed size (to obtain better times, standard compressors must use small blocks, hence achieving poor compression). However, this is basically zero-order compression (i.e., terms are encoded according to their frequency), and hence we are still far from the space usage of, for instance, `lzma`. The goal of the next approach is to maintain the position extraction times of VByte/VNibble, yet achieving higher-order compression.

5.3 Using Natural-Language Compression Boosters

To obtain higher-order compression, we propose to use a so-called natural-language compression booster [22]. The idea is to use first a (hopefully byte-oriented) zero-order compressor on the text (like byte-oriented word Huffman, or even VByte/VNibble on Turpin et al.'s approach). Then this compressed text is further compressed using some standard compression scheme, based on the LZ77 approach (e.g., `lzma`, `snappy` and `lz4`).

Since we will use a standard compressor again, we must organize the text collection in blocks of fixed size, as in Section 5.1. We compress, using a zero-order compressor, consecutive documents, until the accumulated size of the compressed documents surpasses the size defined for blocks. After reaching this condition, the block is recompressed with a high order compressor. To make the decompression process more efficient, we store a table which for each document, stores the block that contains it, as well as its position within the zero-order representation of the block. An illustration of this process can be seen in Figure 5.1.

It has been shown that this can yield better compression ratios than just using a standard compression scheme [22] (especially for smaller block sizes). This is because the zero-order compressor virtually enlarges the LZ77 window (typically of 64 KB), hence more regularities can be detected and compressed. Also, block sizes are defined for the text compressed with the zero-order compressor. In other words, the block that is given as input to the higher-order compressor correspond to a bigger segment of the real text, In our case, we propose using Turpin et al.'s approach [45] as the booster (using VByte and VNibble as we explained above) on the sequence of term identifiers, rather than Word Huffman or End-Tagged as in [22]. Our experiments indicate that the former are faster and use only slightly more space than the latter.

To obtain the positions of a query term within a document D_{i}, as described in Section 5.1, we must first obtain the document and then search for the query terms. To do this, first we have to decompress the block containing D_{i} with the higher-order decompressor. Second, we have to decompress (using the zero-order decompressor) just the part of the block than contains the document needed. Then, the document is ready

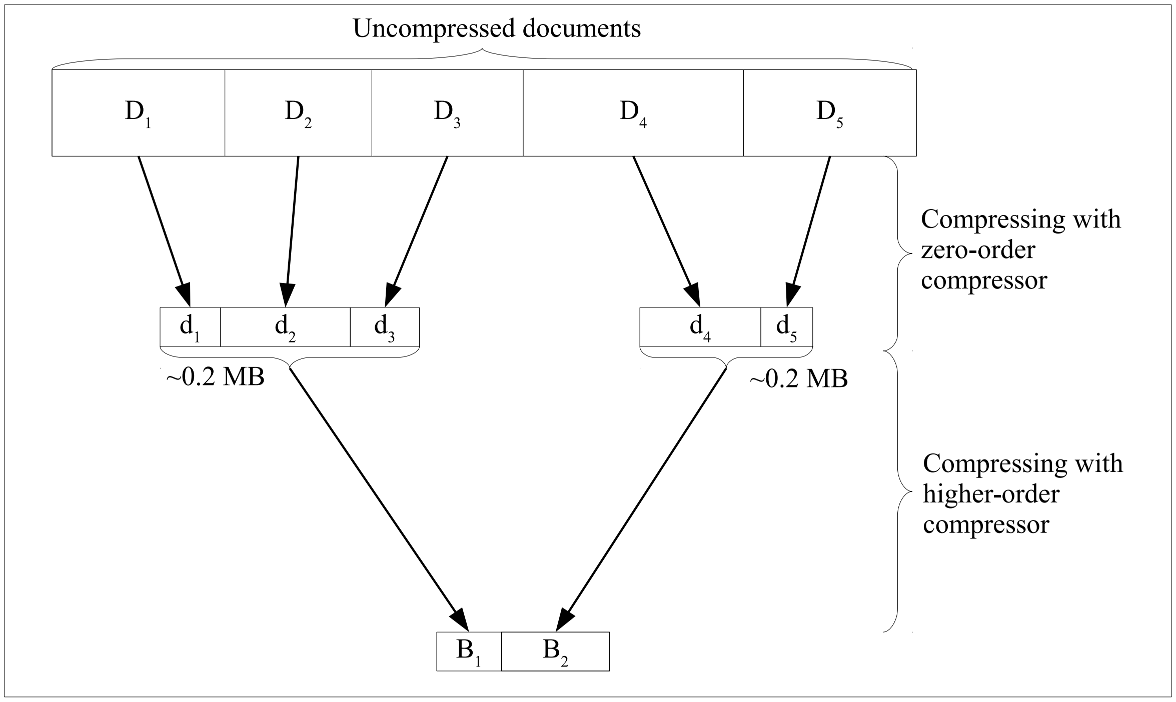


Figure 5.1: The compression boosting scheme, with a block size of 200 KB.

to be searched for the query terms. Figure 5.2 shows this process.

In Table 5.1 (on Page 42) we show results for the compression boosting approaches (see the rows for "VByte + `lzma`", "VByte + `snappy`" and "VByte + `lz4`", for different block sizes). We do not show in the table results for VNibble and byte-oriented Huffman as boosters. This is because our experiments show that using VByte as a booster produces the minimal space usage, compared to that achieved with VNibble and byte-oriented Huffman. That is, even though VNibble and byte-oriented Huffman on their own achieve better space usage than VByte, the combination of VByte plus a higher-order compressor uses from 9% to 16% less space than VNibble as a booster, and from 2% to 6% less space than byte-oriented Huffman as a booster, depending on the higher-order compressor used.

The better performance of VByte as booster compared to VNibble can be explained because VByte is byte aligned, hence higher-order compressors (which are also byte aligned) are able to detect and compress the regularities of the text in a better way. VNibble, on the other hand, is not byte aligned, then many regularities are not detected, and hence the compression ratio achieved is poorer. Finally, byte-oriented Huffman needs to store a data structure for the decoding process (typically, the Huffman tree). This makes Huffman use slightly more space than VByte. Regarding the coding process, VByte can be decoded very efficiently, using fairly simple code, whereas byte-oriented Huffman needs to traverse the Huffman tree to decode.

Comparing now the performance of VByte as booster when used with the different higher-order compressors tested, we can see that, overall, the reduction in space usage (compared to the original VByte approach) is considerable. Comparing VByte + `lzma`

Table 5.1: Experimental results for extracting in-document position data (Step 2) from the document collection. The compression ratio is computed according to the size of the uncompressed text, which is 91,634 MB.

Approach	Compression Scheme	Space (MB)	Compression Ratio	Position extraction time (msec/query)		
				$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
PILs (no text)	S16	31,338	34.19	0.74	1.75	2.51
	RICE	28,373	30.96	1.28	3.27	5.57
Compressed self-indexes	WT(7 KB)	40,534	44.23	1.94	6.85	9.80
	WT(1 KB)	56,917	62.11	0.33	1.15	1.75
	WT(7 KB) + lzma	19,628	21.42	19.25	68.36	97.71
	WT(1 KB) + lzma	42,359	46.23	7.22	24.97	35.57
	WT(7 KB) + snappy	25,122	27.42	14.35	51.02	74.56
	WT(1 KB) + snappy	46,778	51.05	2.07	7.32	10.47
	WT(7 KB) + lz4	24,911	27.18	14.55	51.05	74.60
	WT(1 KB) + lz4	46,600	50.85	2.08	7.31	10.48
Text compressors	lzma (200 KB)	14,987	16.35	137.60	482.09	684.94
	snappy (200 KB)	34,576	37.73	9.47	33.49	47.74
	lz4 (200 KB)	30,405	33.18	6.60	23.22	33.09
Zero-order compressors	VByte	38,339	41.84	0.41	1.40	2.02
	VNibble	34,570	37.73	1.86	6.75	8.10
	Byte-Oriented Huffman	38,070	41.55	1.09	3.82	5.45
Compression boosters	VByte + lzma (200 KB)	12,486	13.63	256.16	906.54	1,284.87
	VByte + lzma (50 KB)	13,981	15.26	70.32	246.94	351.71
	VByte + lzma (10 KB)	16,762	18.29	19.26	68.00	97.04
	VByte + lzma (1 KB)	22,340	24.38	6.11	21.72	31.10
	VByte + snappy (200 KB)	20,158	21.99	9.71	34.01	48.69
	VByte + snappy (50 KB)	20,366	22.22	2.36	8.36	11.95
	VByte + snappy (10 KB)	22,086	24.10	0.82	2.91	4.17
	VByte + snappy (1 KB)	27,919	30.47	0.45	1.60	2.30
	VByte + lz4 (200 KB)	18,361	20.04	6.14	21.64	30.81
	VByte + lz4 (50 KB)	18,845	20.56	1.82	6.44	9.20
	VByte + lz4 (10 KB)	21,665	23.64	0.68	2.42	3.46
	VByte + lz4 (1 KB)	27,680	30.21	0.41	1.43	2.05

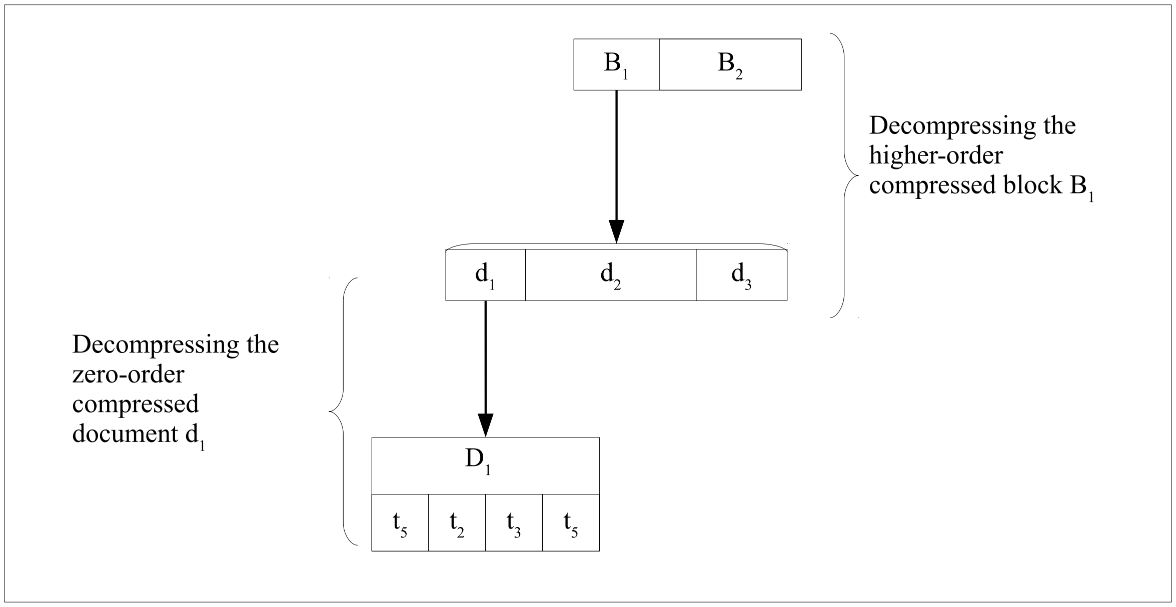

B_1
B_2
Decompressing the higher-order compressed block B_1
d_1
d_2
d_3
Decompressing the zero-order compressed document d_1
D_1
t_5
t_2
t_3
t_5

Figure 5.2: Example of the decompression process for a document D_1 in the compression-boosting scheme.

(200 KB) with `lzma` (200 KB), the result is a reduction in space usage of 16.68% (12,486 MB vs 14,987 MB), but at the cost of twice the running time of the original `lzma`. For VByte + `snappy` (200 KB), on the other hand, we obtain a reduction of 41.69% in space for blocks of size 200 KB against `snappy` (200 KB), with a minor increase in average position-extraction time.

When we use VByte as booster of `lz4`, the size of the structure is reduced compared with `lz4` (for all block sizes), without affecting much the average position-extraction time. Notice also that for all compressors, when using smaller blocks, the time to obtain positions rapidly improves, while the size does not increase considerably in some cases. For example, using a block size of 50 KB with VByte + `snappy`, the average position-extraction time decreases to 2.36 milliseconds per query, which is competitive with the time to obtain positions from `PIL` (Rice). We can conclude that the best alternative is the use of VByte as booster of `lz4`, obtaining (depending on the block size) better average position-extraction time and smaller size than `PIL`s, making both techniques competitive in both space and time. However, VByte + `lz4` also contains the text within this space, allowing its use during snippet generation.

5.4 Further Comparison Between the Most Competitive Alternatives

The most important result from previous sections is that we have found an alternative that is competitive with `PIL`s. Our alternative does not use any index data structure for positions, but just the compressed text. Our results indicate that having an index for positions is not always the best strategy. In this section we will do a more detailed analysis about the behavior of our alternatives. The idea is to find in which cases the alternative of not indexing positions is effective. In particular, we will study the performance of the proposed schemes from different perspectives:

- Space/time trade-offs provided.
- Average position-extraction time as a function of the query length.
- Average position-extraction time as a function of k_1.

Our analysis will be based on alternative VByte + `lz4`, since it was the best performer in previous section. We will use block sizes of 5 KB, 10 KB, 50 KB, 200 KB, 500 KB and 1,000 KB.

5.4.1 Space/Time Trade-Offs

One drawback of `PIL`s is that they have very few parameters that can be tunned to provide space/time trade-offs. Basically, the parameters are the type of compression used (in our case we use Rice) and the chunk size for the `docID` layer. The latter is usually 128, which has shown to be the most effective value. In our setting the minimum space achieved by `PIL`s is 28,373 MB. Hence, if the space allowed for the positional structure is less than that, `PIL`s cannot be used. In the case of compression boosters, the use of larger blocks allows us to achieve a smaller space usage. Indeed, the minimum space achieved is (obviously) when no block structure is used, but the text is compressed as a whole. For VByte + `lzma`, this is 8,394 MB, whereas for VByte + `lz4` it is 17,413 MB. Also, our experiments indicate that using blocks of size bigger than 200 KB only yields little improvements in space usage, of up to 8% for VByte + `lzma` and up to 2% for VByte + `lz4`. This can be seen in Figure 5.3, where we show the experimental space usage as a function of the block size. Moreover, using a block size bigger than 200 KB increases dramatically the average position-extraction time. This is illustrated in Figure 5.4, where it can be seen how (for VByte + `lz4`) the time increases with almost no further improvement in space for blocks of size 500 KB and 1,000 KB

5.4.2 Average Position-Extraction Time as Function of the Query Lengths

In this section we study how the schemes behave for queries of different length (in number of terms). The time needed to extract positions using compression boosters, can be computed as follows. For each document D_{i} in the top-k_1 for a query Q, let $T(B_{D_{\mathrm{i}}})$ be the time needed to obtain the positions of the document D_{i} (which is stored in block $B_{D_{\mathrm{i}}}$). To obtain the positions for the query terms from Q in document D_{i}, we follow the steps described in Section 5.3 (and Figure 5.2). In practice, even when the number of comparisons needed to extract the positions for a decompressed document

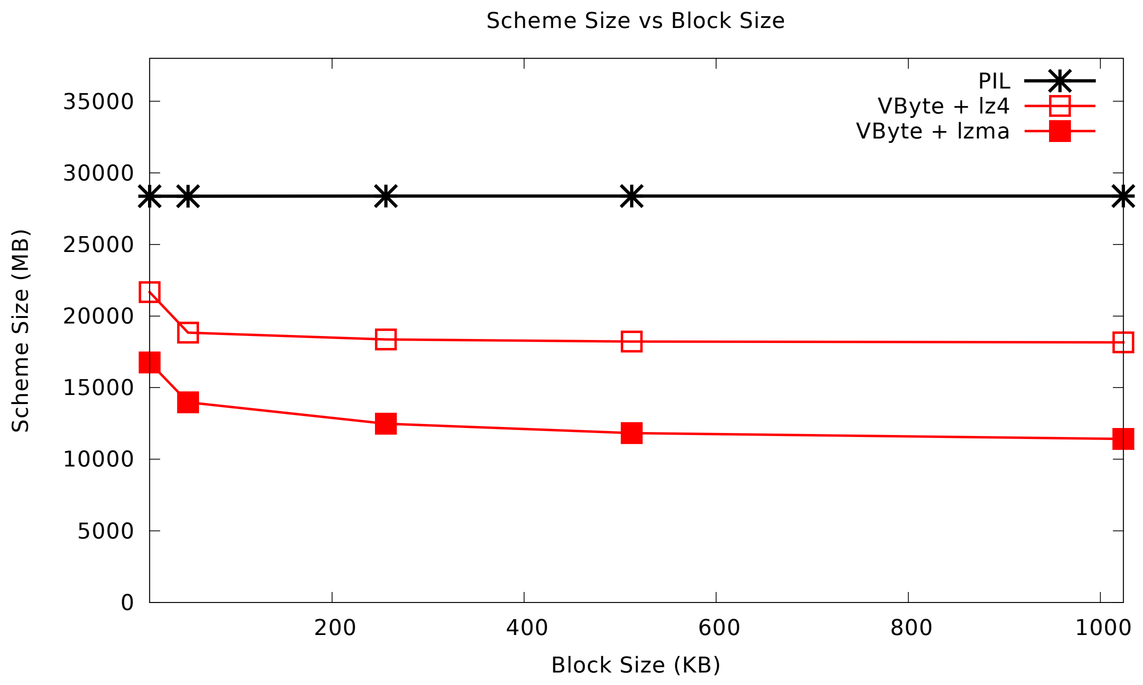


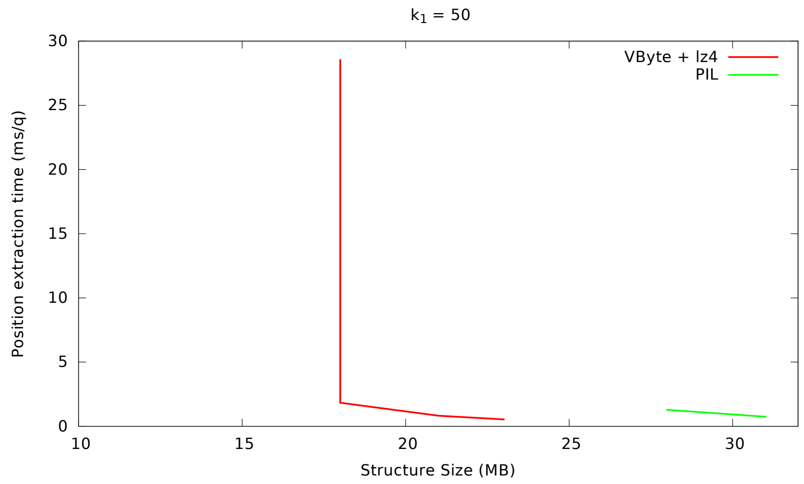
Figure 5.3: Space usage for compression boosters, for different block sizes, and `PIL`s.

D_{i} are $|Q| \cdot |D_{\mathrm{i}}|$, where $|Q|$ is the number of query terms of Q and $|D_{\mathrm{i}}|$ is the number of words in document D_{i}, the time $T(B_{D_{\mathrm{i}}})$ is basically dominated by the time needed to decompress the block $B_{D_{\mathrm{i}}}$, which can be seen as a constant with respect to $|Q|$ (since it just depends on the size of the original block for compression boosters). This can be observed in Figure 5.5, where the average position-extraction remains almost constant as we increase the number of query terms. The worst case of position-extraction time is when every document of the top-k_1 are in a different block. We can conclude that the average position-extraction time depends only on the block size and on k_1.

Due to the high average position-extraction times, for the compression boosters schemes with blocks larger than 50 KB, the following comparison with `PIL`s is done using blocks of size 5 KB, 10 KB and 50 KB.

The time needed to extract positions from `PIL`s can be computed as follows: given a query Q and its top-k_1 results, for each document D_{i} in the top-k_1, we must obtain the positions of the $|Q|$ query terms within D_{i}. This means that in the worst case $k_1 \cdot |Q|$ positional chunks must be decompressed. Hence and unlike the case of compression boosters, the average position-extraction time for `PIL`s depends on $|Q|$. This effect can be observed in Figures 5.6, 5.7 and 5.8, for different values of k_1. As in can be seen, the average position-extraction time for `PIL`s grows lineally with $|Q|$.

To conclude, the increase of query length affects `PIL`s considerably, whereas, compression boosters are not affected at all by this fact.



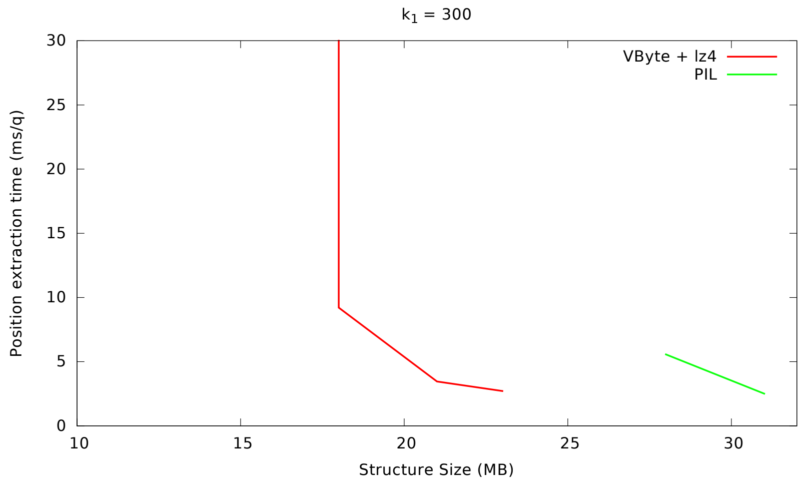


Figure 5.4: Position-extraction time for scheme VByte + `lz4`, for block sizes 5 KB, 10 KB, 50 KB, 200 KB, 500 KB, and 1,000 KB (from right to left in the curve). For comparison, we also show the performance of `PIL`s.

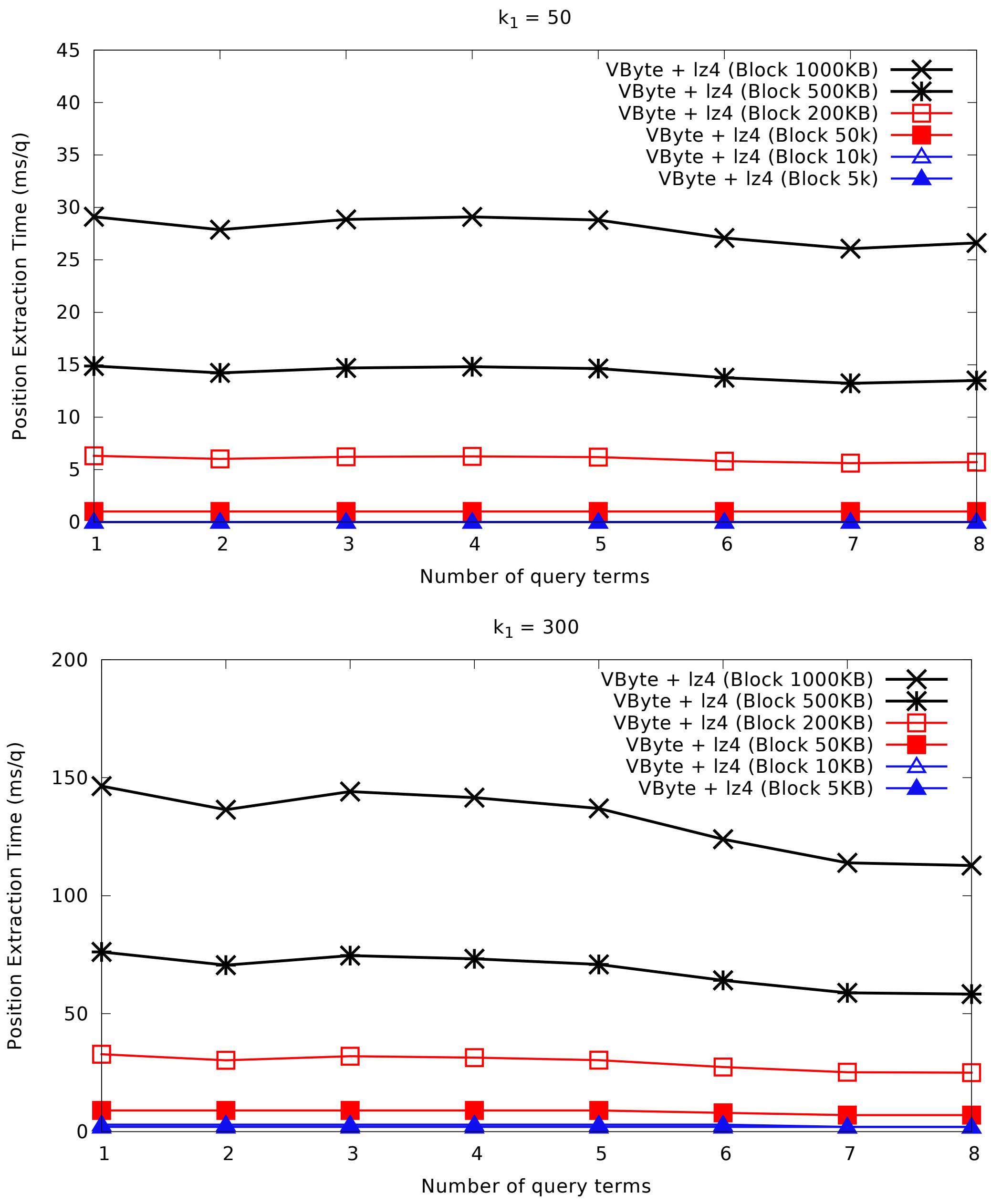


Figure 5.5: Average position-extraction times (for $k_1 \in \{50, 300\}$) per number of query terms, for block sizes of 5 KB, 10 KB, 50 KB, 200 KB, 500 KB, 1,000 KB.

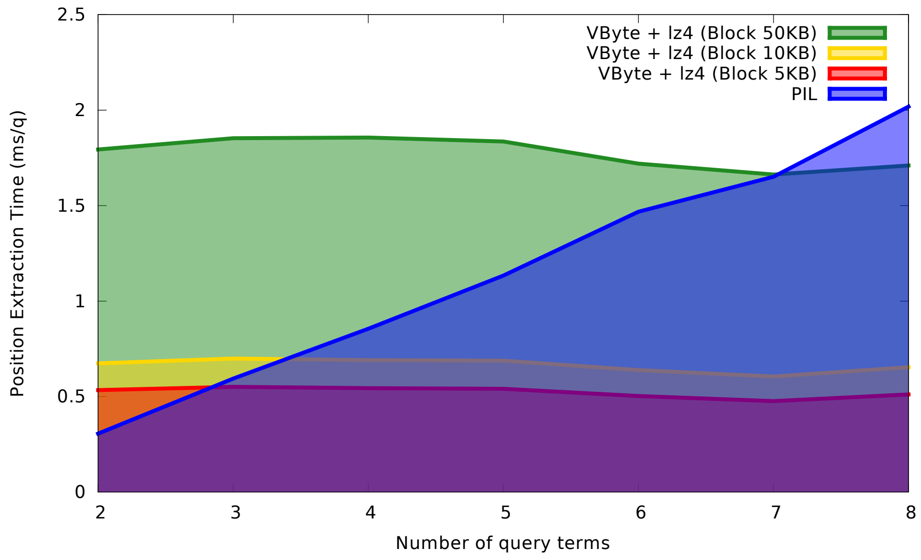


Figure 5.6: Average position-extraction time for compression boosters and PIL, for $k_1 = 50$.

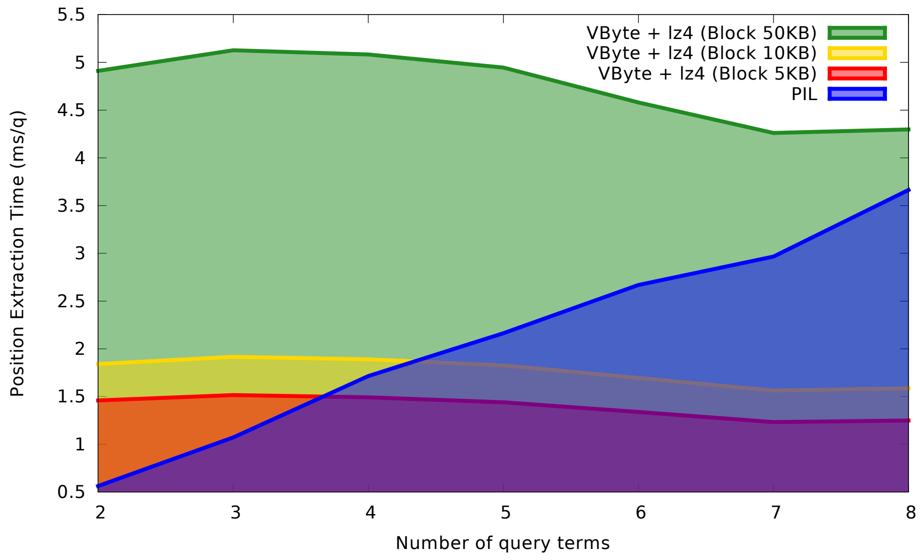


Figure 5.7: Average position-extraction time for compression boosters and PIL, for$k_1 = 150$.

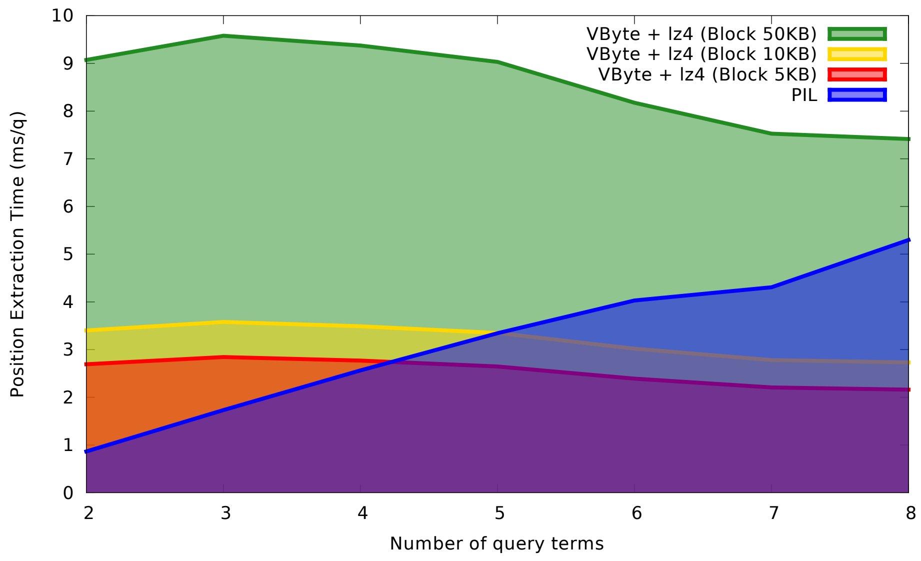


Figure 5.8: Average position-extraction time for compression boosters and PIL, for $k_1 = 300$.

5.4.3 Average Position-Extraction Time as Function of k_1

In this section we show the behavior of the schemes as the value of k_1 varies. Figure 5.9 shows that the average position-extraction time of both schemes is lineal in k_1. Notice that the growth for PILs as k_1 increases, is the slowest. This is because as k_1 grows, many documents in the top-k_1 are within the same chunk in the inverted index, hence the cost of decompressing these chunks grows slowly. Text blocks, on the other hand, store much less documents (tens of them), so the probability of decompressing k_1 text blocks is high (which we also observed on our experiments). Moreover, as we increase the block size, more unuseful documents are decompressed, which explains why the cost for blocks of 50 KB grows faster.

5.5 Conclusions

From our experiments we can conclude that our alternative VByte + lz4 is a very good choice to support positional ranking and snippet generation. If we consider the query lengths, our experiments indicate that PILs are more adequate for short queries. For long queries, on the other hand, the number of inverted lists involved makes PILs less competitive. In such a case, VByte + lz4 should be preferred. Long queries appear in many relevant applications, such as search engines for targeted advertising (where queries contain many terms involving the characterization of a user, for instance). Finally, we conclude that approach VByte + lz4 is more sensitive to the value of k_1 than PILs, in the sense that the position-extraction time for the former degrades faster than that of PILs when the value of k_1 is increased.

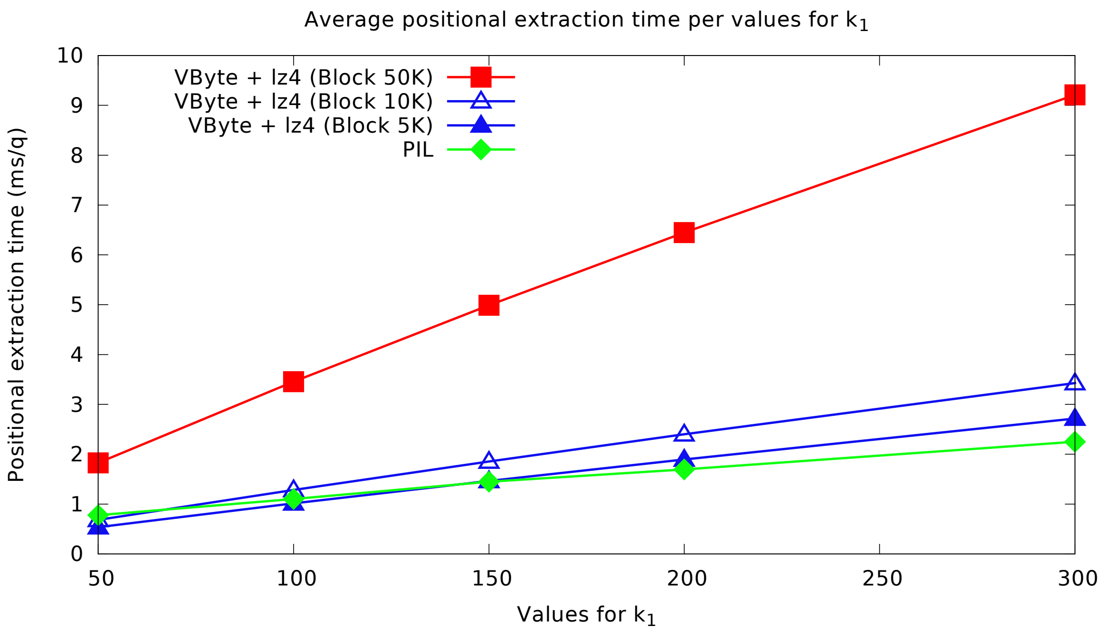


Figure 5.9: Comparison between compression boosters and PILs, for different values of k_1.

Chapter 6

Discussion and Further Experimental Results

In this chapter we consider the best indexing alternatives from previous chapters to build schemes that allow us to carry out the search process as described in Section 3.1 (on page 24). The idea is to show the available trade-offs for the complete search process.

6.1 Scenario 1: Query Processing with Snippet Generation

The first scenario we test implements the three steps from Section 2.1 (i.e., query processing, positional ranking step, and snippet generation). In this scenario we must store the document collection, and must be able to compute position data. Latter in this chapter we will study an alternative scenario where snippets are not needed, hence the text is not needed. Table 6.1 defines the schemes used in our experiments. All schemes include the inverted index to carry Step 1 in the query process. The inverted index includes `docIDs` (compressed with `PforDelta`) and frequencies (compressed with S16). The total space usage for the inverted index for GOV2 collection is 9,739 MB. For the query process, we set $k_1 \in \{50, 100, 150, 200, 300\}$ and $k_2 \in \{10, 30, 50\}$. As in the experiments of previous chapters, we use the TREC 2006 query log. In order to compare and understand how the different position extraction schemes impact on the overall query time, we show in each case the time needed for Step 1 of the query processing (in our case, for "AND" and "BMW OR" query processing).

The space/time trade-offs for the schemes tested are determined as follows:

lzma: We used text blocks of size 200 KB, 500 KB and 1000 KB.

lz4: We used text blocks of size 200 KB, 500 KB and 1000 KB.

WT: We used text blocks of size 1 KB, 2 KB and 7 KB.

WT + lzma: We used text blocks of size 1 KB, 2 KB and 7 KB.

WT + lz4: We used text blocks of size 1 KB, 2 KB and 7 KB.

VByte/VNibble: The two points in the trade-off for this case are obtained by using either VNibble compression for the text (the least space) and VByte compression for the text.

VByte + lzma: We used text blocks of size 5 KB, 10 KB, 50 KB and 200 KB.

VByte + lz4: We used text blocks of size 5 KB, 10 KB, 50 KB and 200 KB.

PIL + lzma: We used text blocks of 200 KB (for `lzma`). The two points in the trade-off are obtained using Rice compression for the `PIL`s (the least space) and S16.

PIL + lz4: We used text blocks of 200 KB (for `lz4`). The two points in the trade-off are obtained using Rice compression for the PIL s (the least space) and S16.

PIL + VNibble: The two points in the trade-off are obtained using Rice compression for the `PIL`s (the least space) and S16.

PIL + VByte + lzma: The two points in the trade-off are obtained using Rice compression for the `PIL`s (the least space) and S16. For the compressor booster we used text blocks of size 50 KB.

PIL + VByte + lz4: The two points in the trade-off are obtained using Rice compression for the `PIL`s (the least space) and S16. For the compressor booster we used text blocks of size 50 KB.

Table 6.1: Glossary of the indexing schemes tested. All schemes include the inverted index, which for the GOV2 collection represents 9,739 MB.

Indexing Scheme	Description
`lzma`	Text compressed with `lzma` for positions and text.
`lz4`	Text compressed with `lz4` for positions and text.
`WT`	`WT` for positions and text.
`WT`+ `lzma`	`WT` compressed with `lzma` for positions and text.
`WT`+ `lz4`	`WT` compressed with `lz4` for positions and text.
VByte/VNibble	Text compressed with VByte/VNibble for positions and text.
VByte + `lzma`	VByte compression booster on `lzma` for positions and text.
VByte + `lz4`	VByte compression booster on `lz4` for positions and text.
`PIL`+ `lzma`	`PIL`s for positions, text compressed with `lzma`.
`PIL`+ `lz4`	`PIL`s for positions, text compressed with `lz4`.
`PIL`+ VNibble	`PIL`s for positions, text compressed with VNibble.
`PIL`+ VByte + `lzma`	`PIL`s for positions, text compressed with VByte + `lzma`.
`PIL`+ VByte + `lz4`	`PIL`s for positions, text compressed with VByte + `lz4`.

6.1.1 DAAT AND Queries

We test first DAAT AND queries for Step 1. This kind of queries are usually supported very efficiently, and correspond just to a small fraction of the overall query process. This means that the time needed to obtain position data should be also efficient, and small differences among alternatives could influence the total processing time.

For $k_1 = 50$ and $k_2 = 10$ (Figure 6.1, upper plot), it can be seen that schemes that use `PIL`s to index position data have a high space usage, as they need to store the text to obtain snippets. Some of these schemes offer a competitive query time, yet their space usage makes them unaffordable. `PIL` + VByte + `lz4` is the most space-efficient alternative using `PIL`s, yet it cannot compete at query time. This shows in practice what we have mentioned along this thesis: state-of-the-art solutions for positional data have a high space usage. This is because they need to store both positions and text separately, and in particular positions are not as compressible as other index components (recall that even the text is more compressible than position).

Scheme `WT` offers also a highly competitive query time, yet using less space than schemes using `PIL`s. Yet the space usage is still high. Alternatives `WT` + `lzma` and `WT` + `lz4` yield a significant reduction of about 42% in space usage. However, the resulting query time degrades quickly, and becomes less competitive.

Scheme VByte/VNibble (which implements Turpin et al.'s idea [45] to compress the text) yields a good trade-off, with small space usage than `WT` and a highly-competitive query time. Up to this point we have achieved a reduction of about 21% in space usage compared to the most time-efficient alternative that uses `PIL`s (i.e., the state of the art up to now), with almost the same query time. However, the space usage is still high compared to the alternatives we study next.

Scheme `lz4` reduces the space usage even more, providing a competitive query time. Scheme `lzma`, on the other hand, reduces the space usage significantly, yet at the price of a much slower (unpractical) query time.

When we add the compressor boosters to `lzma` and `lz4` (i.e., schemes VByte + `lzma` and VByte + `lz4`), we obtain the most important trade-offs among all alternatives. For instance, if we compare with `PIL` + `lz4` (which is the fastest alternative in the state of the art, and is among the most space-efficient alternatives that use `PIL`), VByte + `lz4` (with text block of size 50 KB, which is the third point in the curve from the right) yields a significant reduction in space usage of about 49.81% (56,958 MB versus 28,584 MB), with a query time that is just 1.03 times slower (from 16.95 msecs/query to 17.49 msecs/query). This shows the effectiveness of our proposal. Moreover, recall that the positional index `PIL` (using Rice compression) requires 28,373 MB, whereas VByte + `lz4` requires, as we already said, 28,584 MB. That is, in 1.007 times the space of just `PIL`s we can store the inverted index (`docID`s and frequencies), the positional data, and the text collection. In other words, using only slightly more space than `PIL`(Rice), scheme VByte + `lz4` includes everything needed for query processing. This is one of the most important results and conclusions in this thesis: "not to index" positional data can be a highly-competitive alternative in practical scenarios.

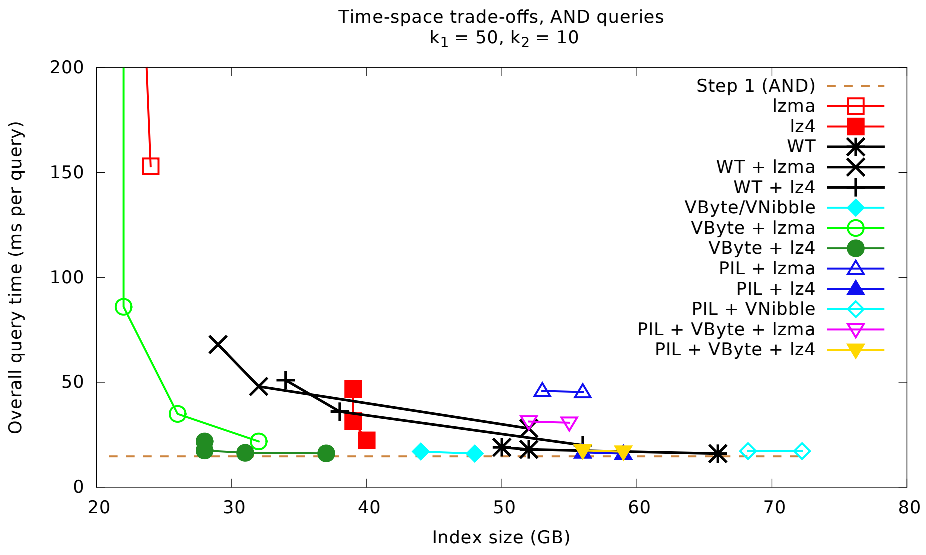
If we compare now `PIL`+ `lz4` with VByte + `lzma` (for blocks of size 10 KB, which corresponds to the second point in the curve, from the right), we obtain a reduction in space usage of about 53.47% (56,958 MB versus 26,501 MB), while the query time is 1.99 times slower (from 17.47 msecs/query to 34.93 msecs/query). In other words, we are able to accommodate the complete index in 0.94 times the space of `PIL`s, and still are able to offer a competitive query time.

Finally, the smallest space alternatives we tested (which are not fully shown in the figures) are the ones that use the inverted index for query processing and `lzma` compression for positions and snippets. This achieves about 22,225 MB of space. This scheme includes everything needed for query processing, and uses only 78% the space of `PIL`. However, query processing time increases significantly, to more than 400 ms per query. This scheme could be useful in some cases where the available memory space is very restricted, such that a larger index would mean going to disk.

A recent alternative [42] proposes to use *flat positional indexes* [15, 18] to support phrase querying; this index could also be used for positional ranking. This is basically a positional index from which `docID` and frequency information can also be obtained. The results reported for the GOV2 collection in [42] give an index of size 30,310 MB that includes `docID`s and frequencies, but not the text needed for snippet generation, making this approach uncompetitive for our scenario.

If we increase k_2 to 50, the result is almost the same (see Figure 6.1, below). The only schemes that are affected (i.e., the query time is increased) are those using `PIL`s (because more documents than before must be decompressed per query, in order to obtain their snippets) and those using `WTs` (as we need to show snippets for more documents, this means that we need to obtain more words from the text, which means traversing the WT more intensively). Schemes that use the compressed text for snippets and positions must decompress k_1 documents to obtain position data, to then extract snippets for the already decompressed top-k_2 documents (using negligible extra space).

Figures 6.1 and 6.2 show experimental results for values of $k_1 \in \{50, 300\}$ and $k_2 \in \{10, 50\}$. Experiments for $k_2 = 30$ and $k_1 \in \{100, 150, 200\}$ are shown in Appendix A. As it can be seen, the same conclusions can be drawn.



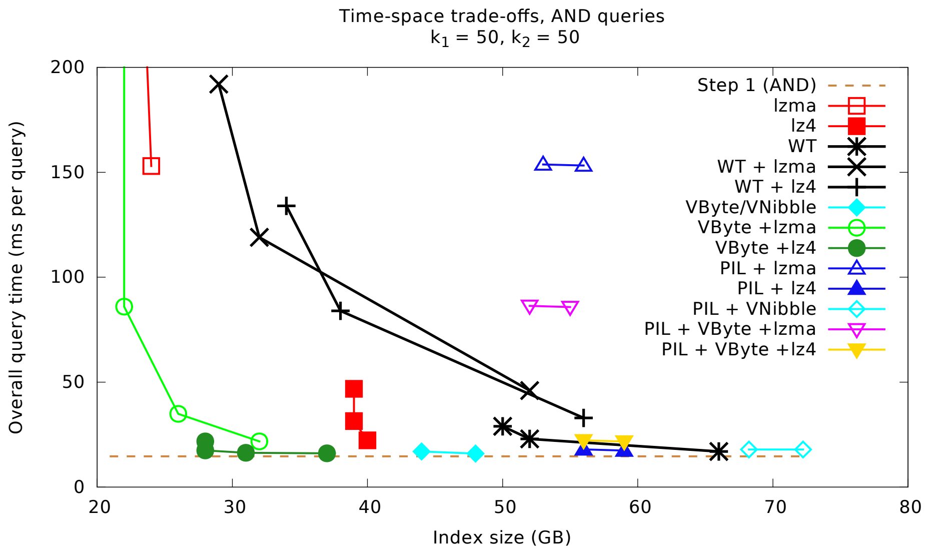
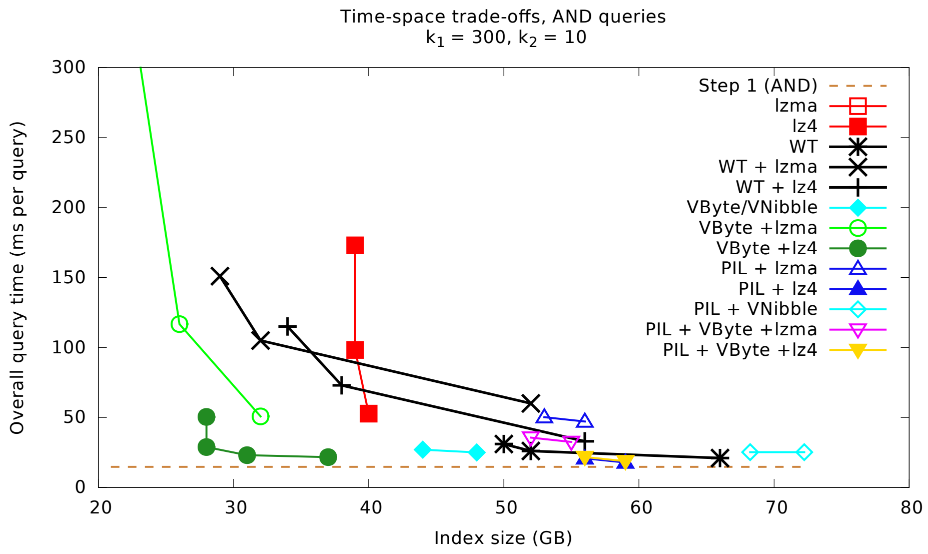


Figure 6.1: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$ and $k_2 \in \{10, 50\}$, including Step 1(AND queries), Step 2 and Step 3.



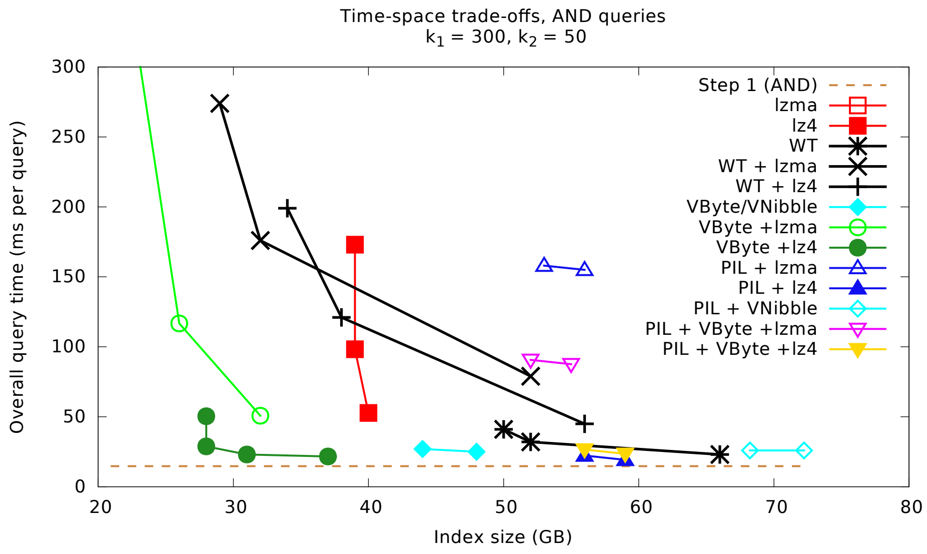


Figure 6.2: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$ and $k_2 \in \{10, 50\}$, including Step 1(AND queries), Step 2 and Step 3. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

6.1.2 BMW OR Queries

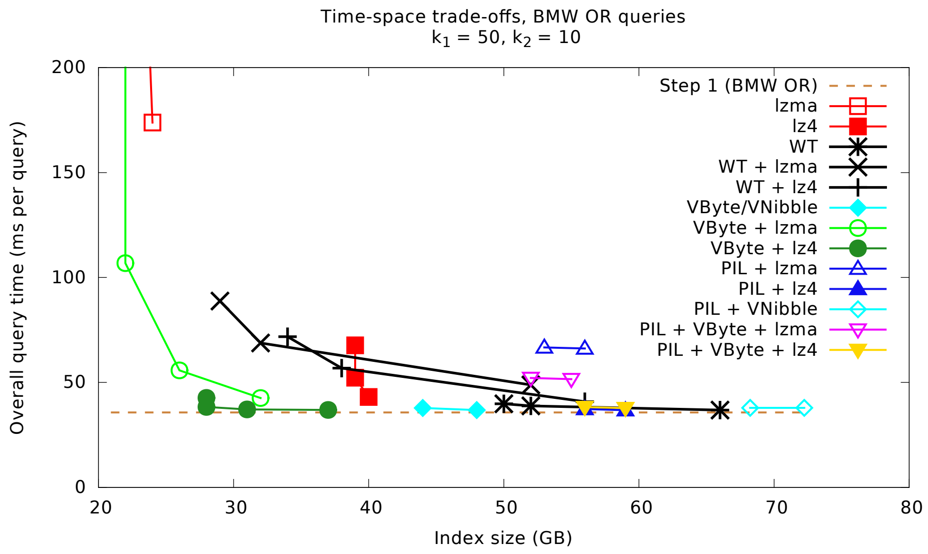
Figures 6.3 and 6.4 show experimental results for BMW OR queries. The results now are slightly different, since now Step 1 of query processing is more expensive than for AND queries. Hence, the differences in query time are smaller. For instance, for $k_1 = 50$ and $k_2 = 10$ (Figure 6.3, upper plot) we can conclude that scheme VByte + `lzma` with text blocks of size 10 KB uses 0.94 times (as we already said) the `PIL` size (which just stores positional data). The query time is 1.43 times slower than that of scheme PILs + `lz4`. For scheme VByte + `lz4`, the results are even more impressive. Using just 1.007 times the space of just PILs we can achieve a query time that is 1.005 times slower than scheme `PIL`+ `lz4`. As in previous section, we conclude that VByte + `lz4` and VByte + `lzma` (using text blocks of size 50 KB) offer very relevant trade-offs, being able to replace `PIL`s in many cases. Experiments for $k_2 = 30$ and $k_1 \in \{100, 150, 200\}$ are shown in Appendix A.

6.2 Scenario 2: Query Processing without Snippet Generation

We now study the practical performance of all proposed schemes in a scenario where text snippets are not needed. That is, Step 3 of Section 3.1 is not carried out. Notice that the text is not needed in the state-of-the-art schemes based on `PIL`s. Hence, we now have just one scheme using `PIL`s (in previous section the various schemes with `PIL`s only differed in how they compressed the text). See Table 6.2 for a summary of the alternatives we tested. In our experiments we set $k_1 \in \{50, 100, 150, 200, 300\}$. All schemes include the inverted index, as in previous section. In order to compare and understand how the different position extraction schemes impact on the overall query time, we show in each case the time needed for Step 1 of the query processing (in our case, for “AND” and “BMW OR” query processing).

6.2.1 DAAT AND Queries

Figures 6.5 and 6.6 show results for AND queries, for values of $k_1 \in \{50, 300\}$, the rest are shown in Appendix A. The most important result to highlight is that the state-of-the-art scheme based on `PIL`s (which this time does not store the text, saving considerable space) cannot compete with schemes VByte + `lz4` and VByte + `lzma`, neither in space usage nor query time. That is, "not to index" positional data is also effective in scenarios where snippets are not needed.



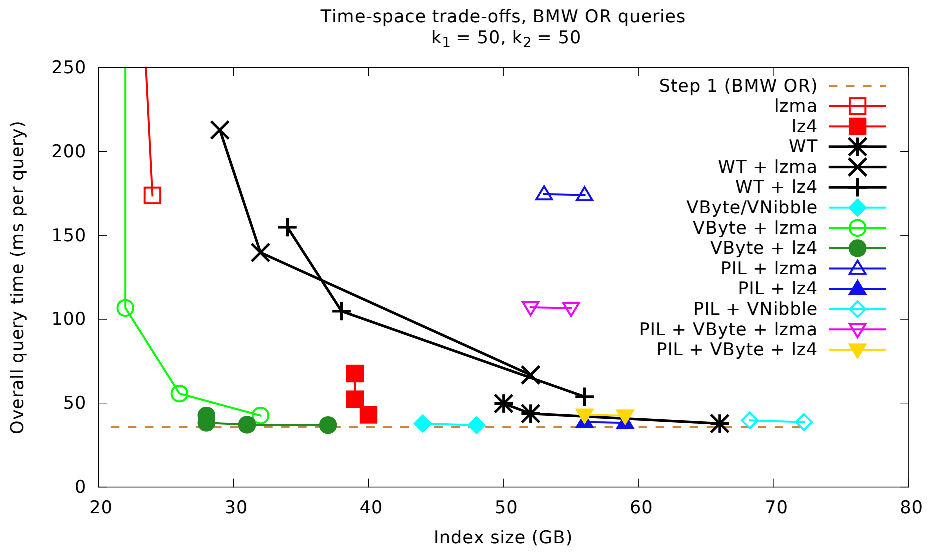
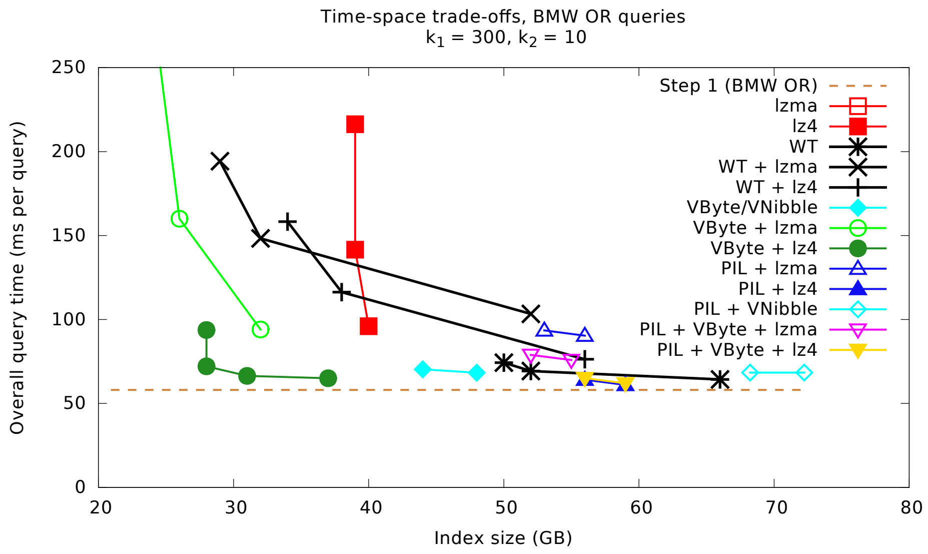


Figure 6.3: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$ and $k_2 \in \{10, 50\}$, including Step 1(BMW OR queries), Step 2 and Step 3.



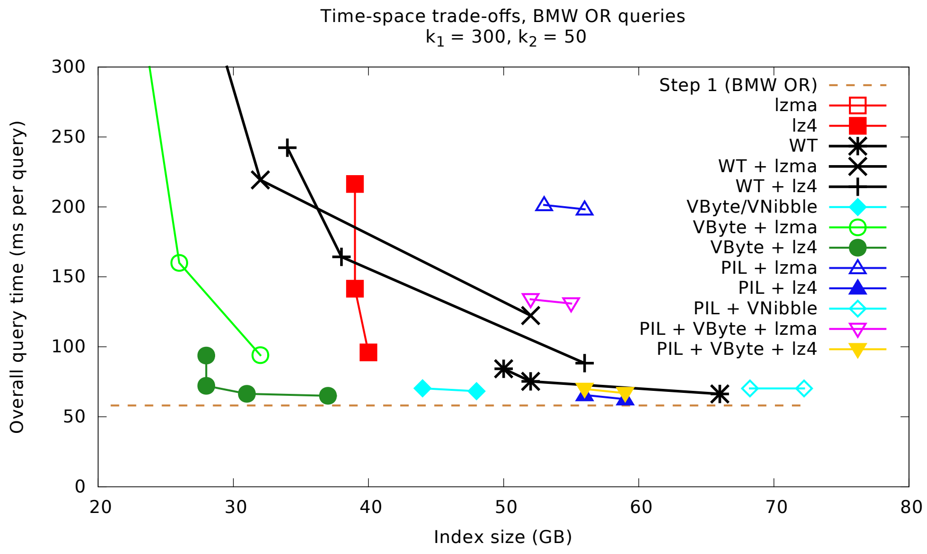


Figure 6.4: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$ and $k_2 \in \{10, 50\}$, including Step 1(BMW OR queries), Step 2 and Step 3. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

Table 6.2: Glossary of the indexing schemes for the figures. All schemes include the inverted index, which for the GOV2 collection represents 9,739 MB.

Indexing Scheme	Description
`lzma`	Text compressed with `lzma` for positions.
`lz4`	Text compressed with `lz4` for positions.
`WT`	`WT` for positions.
`WT`+ `lzma`	`WT` compressed with `lzma` for positions.
`WT`+ `lz4`	`WT` compressed with `lz4` for positions.
VByte/VNibble	Text compressed with VByte/VNibble for positions.
VByte + `lzma`	VByte compression booster on `lzma` for positions.
VByte + `lz4`	VByte compression booster on `lz4` for positions.
`PIL`	`PIL`s for positions.

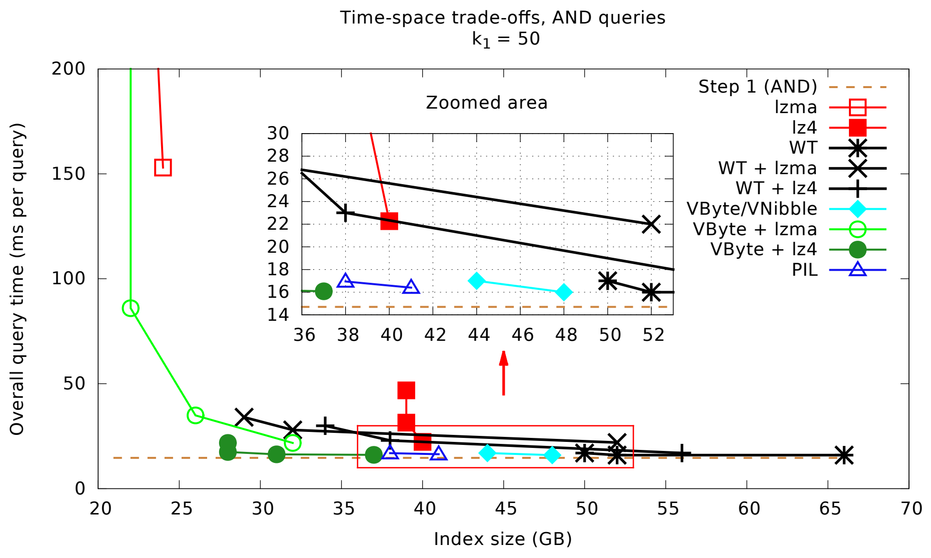


Figure 6.5: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$, including Step 1(AND queries), Step 2.

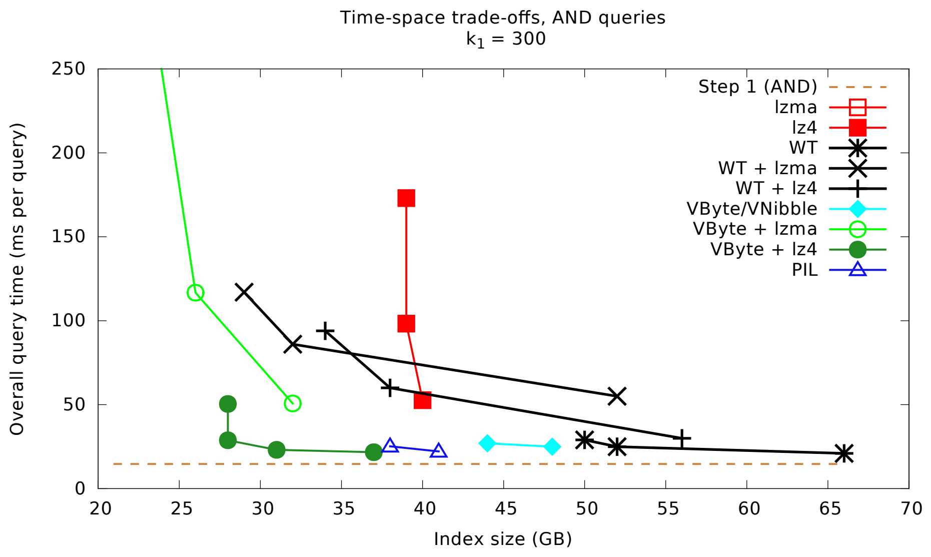


Figure 6.6: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$, including Step 1(AND queries), Step 2. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

6.2.2 BMW OR Queries

Figures 6.7 and 6.8 show results for BMW OR queries, for values of $k_1 \in \{50, 300\}$, the rest are shown in Appendix A. Notice that the differences are even smaller than in Section 6.1.2, because in this scenario the text is not needed, then `PIL`s are not affected by the snippet extraction time. Using the same scenario as in Section 6.1.2, where the scheme VByte +`lz4` is 1.007 times the space of just `PIL`s we can achieve a query time just 1.01 times slower than the `PIL`+ `lz4` alternative. Concluding that "not to index" positional data is effective in all the scenarios studied.

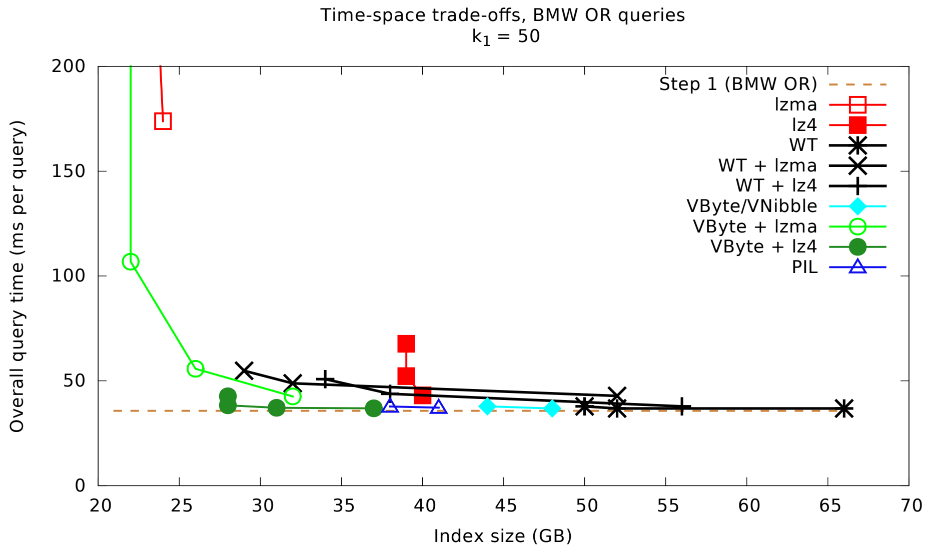


Figure 6.7: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$, including Step 1(BMW OR queries), Step 2.

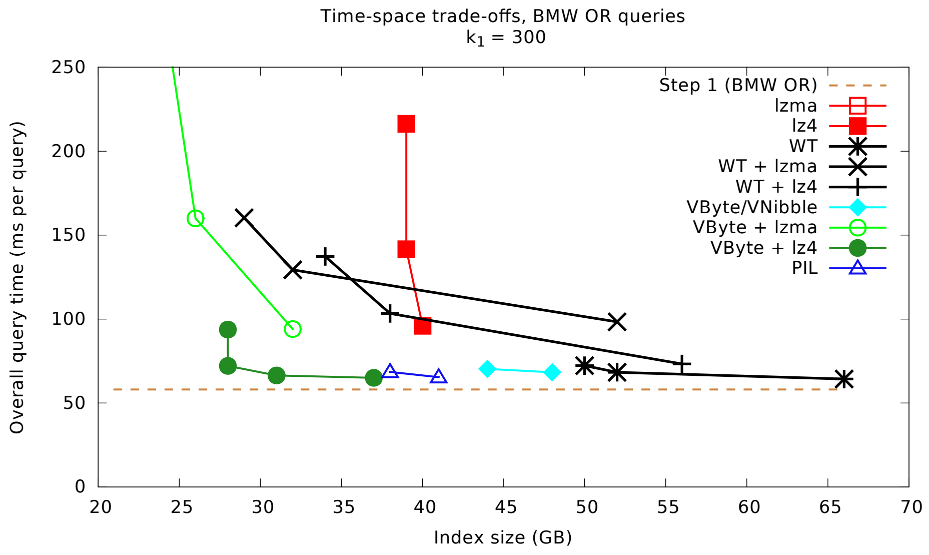


Figure 6.8: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$, including Step 1(BMW OR queries), Step 2. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

Chapter 7

Conclusion and Future Work

In this thesis we have studied alternative approaches for indexing in-document positions. This is relevant for search engines that use the query-term positions within the documents to ranking their results and for phrase searching (though the studies in this thesis focused particularly on positional ranking). It has been shown in the literature that indexing positions is usually a difficult task [49, 25], mainly because of the high space usage they impose. This is because of two things: we must store all positions of a term within each document and, moreover, the way in which positions are usually stored and indexed does not allow one to catch some regularities that could be used to compress this kind of data.

Our main proposal was to get rid of positional indexes, and use the textual data instead to obtaining in-document positions. We tried several approaches to obtain good space/time trade-offs. We also extended the work of [25], in the sense that many other compression alternatives were tried, not just standard compressors.

From our study we can conclude that there exists a wide range of practical time/space trade-offs, other than just the classical positional inverted indexes. We studied several alternatives, trying to answer the question whether it is necessary to index position data or not. As one of the most relevant points in the trade-off, we propose a compressed document representation based on the approach in [45] combined with `lz4` compression [2]. This allows us to compute position and snippet data using less space than a standard positional inverted index that only stores position data. Even if we include the space used for document identifiers and term frequencies, this approach uses just 1.007 times the space of a positional inverted index (that stores only in-document positions), with basically the same query time.

We also found out that interesting results, such as that obtaining positions from the textual data is more effective for long queries.

This means that in many practical cases, “not to index” position data may be the most efficient approach. This provides new practical alternatives for positional index compression, a problem that has been considered difficult to address in previous work

[49, 25]. Finally, we also showed that compressed self-indexes such as wavelet trees [27] can be competitive with the best solutions in some scenarios.

We think that our results will change the way in which positional data in web search engines is indexed. However, there are still aspects that need further study in order to obtain improvements. For instance, an interesting line of research could be that of reducing the number of text blocks that are decompressed at query time. This means finding a way to group the documents such that, given a query, most of the top-k_1 documents for the query are within the same (or a few) block. It is interesting also to study what happens in document collections that can be renumbered [50, 7]. In such a case, it is well known that fewer inverted-list blocks are accessed, hence the performance of positional inverted indexes could be improved. It would be interesting to study how does this affect the document collection and the way positional data is obtained from it. Finally, it would be also interesting to study the efficiency of our approach for phrase searching. This seems to be a promising line of research.

Bibliography

[1] `http://code.google.com/p/snappy/`.

[2] `https://code.google.com/p/lz4/`.

[3] Vo Ngoc Anh and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *SIGIR*, pages 290–297, 1998.

[4] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[5] Diego Arroyuelo, Veronica Gil Costa, Senén González, Mauricio Marín, and Mauricio Oyarzún. Distributed search based on self-indexed compressed text. *Inf. Process. Manage.*, 48(5):819–827, 2012.

[6] Diego Arroyuelo, Senén González, and Mauricio Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *SPIRE*, pages 43–54, 2010.

[7] Diego Arroyuelo, Senén González, Mauricio Oyarzún, and Victor Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *SIGIR*, pages 173–182, 2013.

[8] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[9] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[10] Nieves R. Brisaboa, Ana Cerdeira-Pena, Gonzalo Navarro, and Oscar Pedreira. Ranked document retrieval in (almost) no space. In *SPIRE*, pages 155–160, 2012.

[11] Nieves R. Brisaboa, Antonio Fariña, Susana Ladra, and Gonzalo Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Inf. Retr.*, 15(6):527–557, 2012.

[12] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*,

pages 426–434, 2003.

[13] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.

[14] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *SIGIR*, pages 621–622, 2006.

[15] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *Comput. J.*, 38(1):43–56, 1995.

[16] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, pages 176–187, 2008.

[17] D. Cutting. Apache Lucene. `http://lucene.apache.org/`.

[18] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, page 1, 2009.

[19] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, pages 993–1002, 2011.

[20] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[21] Antonio Fariña, Nieves R. Brisaboa, Gonzalo Navarro, Francisco Claude, Ángeles S. Places, and Eduardo Rodríguez. Word-based self-indexes for natural language text. *ACM Trans. Inf. Syst.*, 30(1):1, 2012.

[22] Antonio Fariña, Gonzalo Navarro, and José R. Paramá. Boosting text compression with word-based statistical encoding. *Comput. J.*, 55(1):111–131, 2012.

[23] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009.

[24] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[25] Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *WSDM*, pages 391–400, 2010.

[26] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[27] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[28] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*,

46(5):604–632, 1999.

[29] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[30] Giovanni Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[31] G Nigel N Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, 1979.

[32] Donald Metzler and W. Bruce Croft. A markov random field model for term dependencies. In *SIGIR*, pages 472–479, 2005.

[33] Gilad Mishne and Maarten de Rijke. Boosting web retrieval through query operations. In *ECIR*, pages 502–516, 2005.

[34] Alistair Moffat. Word-based text compression. *Softw., Pract. Exper.*, 19(2):185–198, 1989.

[35] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[36] Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[37] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[38] Yves Rasolofo and Jacques Savoy. Term proximity scoring for keyword-based retrieval systems. In *ECIR*, pages 207–218, 2003.

[39] David Salomon. *Data compression - The Complete Reference, 4th Edition.* Springer, 2007.

[40] Ralf Schenkel, Andreas Broschart, Seung won Hwang, Martin Theobald, and Gerhard Weikum. Efficient text proximity search. In *SPIRE*, pages 287–299, 2007.

[41] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, pages 222–229, 2002.

[42] Dongdong Shan, Wayne Xin Zhao, Jing He, Rui Yan, Hongfei Yan, and Xiaoming Li. Efficient phrase querying with flat position index. In *CIKM*, pages 2001–2004, 2011.

[43] B. Sparrow, J. Liu, and M. Wegner. Google effects on memory: Cognitive consequences of having information at our fingerprints. *Science*, 333(6043):776–778, 2011.

[44] Tao Tao and ChengXiang Zhai. An exploration of proximity measures in informa-

tion retrieval. In *SIGIR*, pages 295–302, 2007.

[45] Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. Fast generation of result snippets in web search. In *SIGIR*, pages 127–134, 2007.

[46] Lidan Wang, Jimmy J. Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *SIGIR*, pages 105–114, 2011.

[47] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.

[48] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[49] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *SIGIR*, pages 147–154, 2009.

[50] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[51] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[52] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[53] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[54] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.

Appendix A

Aditional Experimental Results

In this section we show the trade-offs for the alternatives described in Table 6.1 with $k_1 \in \{100, 150, 200\}$ and $k_2 = 30$, also the alternatives described in Table 6.2 with $k_1 \in \{100, 150, 200\}$.

A.1 DAAT AND Queries

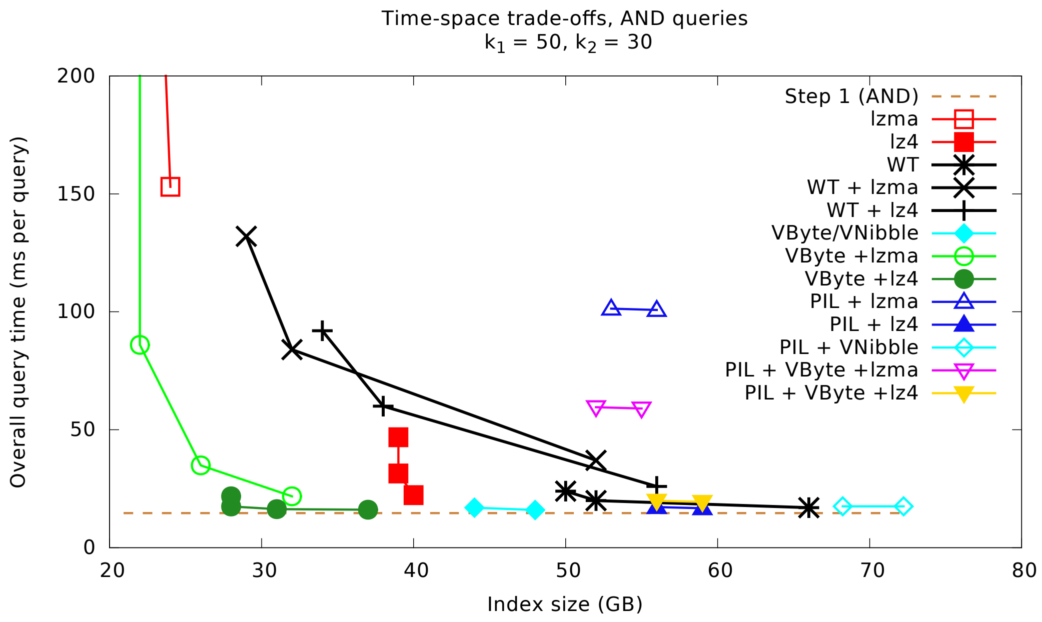
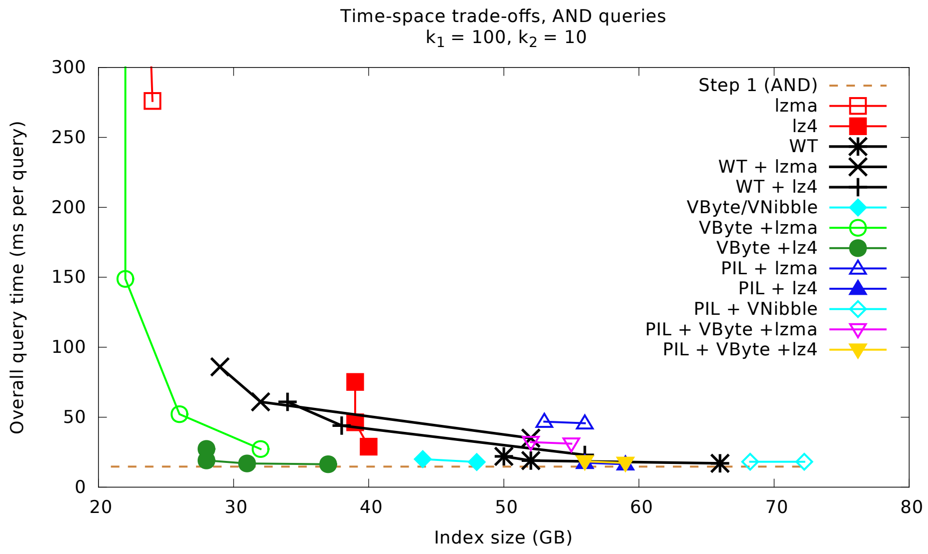


Figure A.4: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$ and $k_2 = 30$, including Step 1, Step 2 and Step 3.



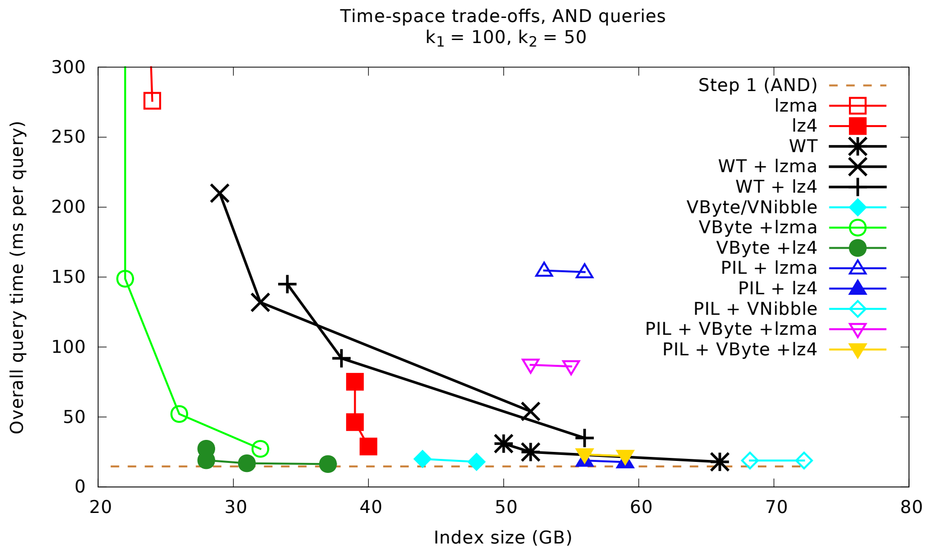
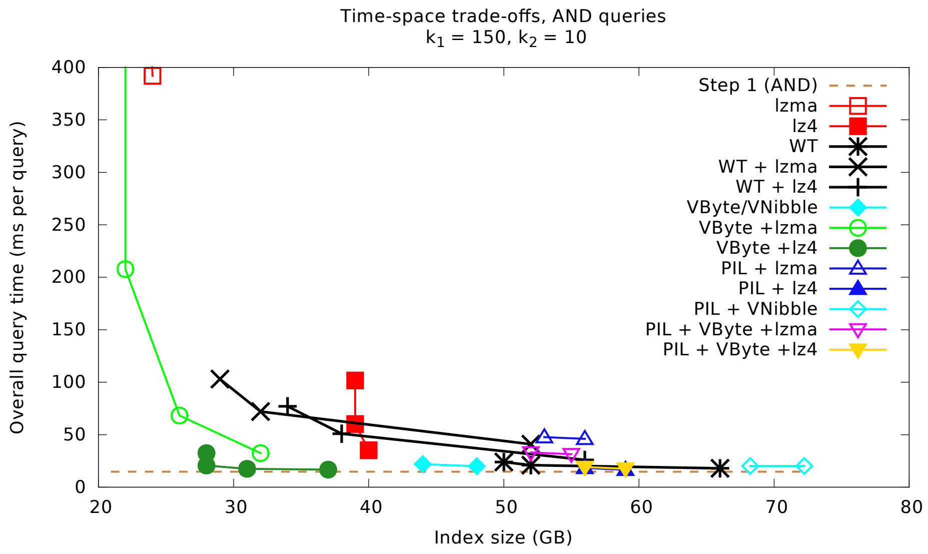


Figure A.1: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$ and $k_2 \in \{10, 50\}$, including Step 1(AND queries), Step 2 and Step 3.



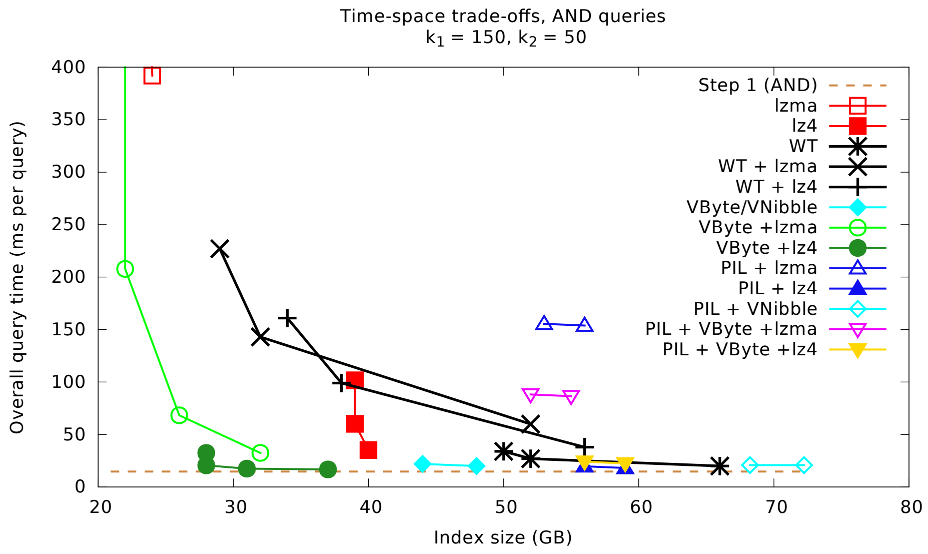
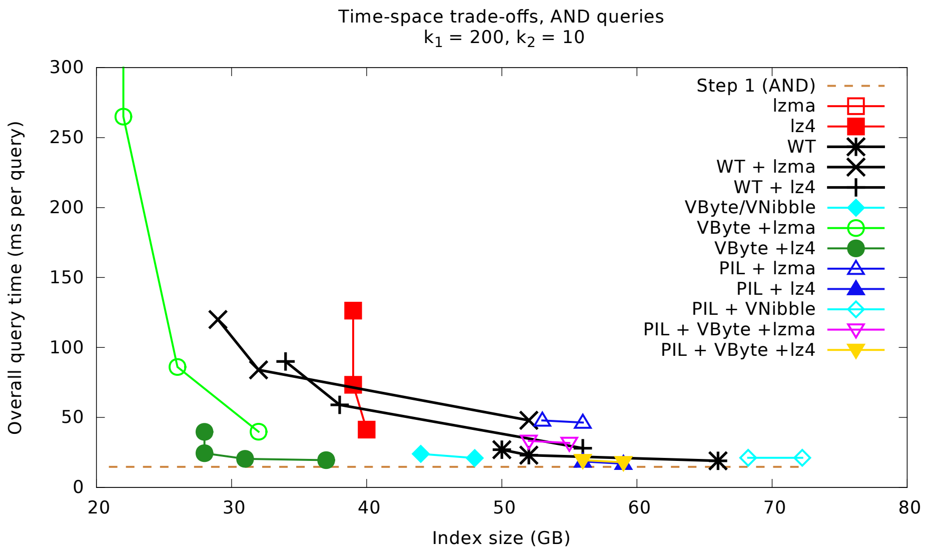


Figure A.2: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$ and $k_2 \in \{10, 50\}$, including Step 1(AND queries), Step 2 and Step 3.



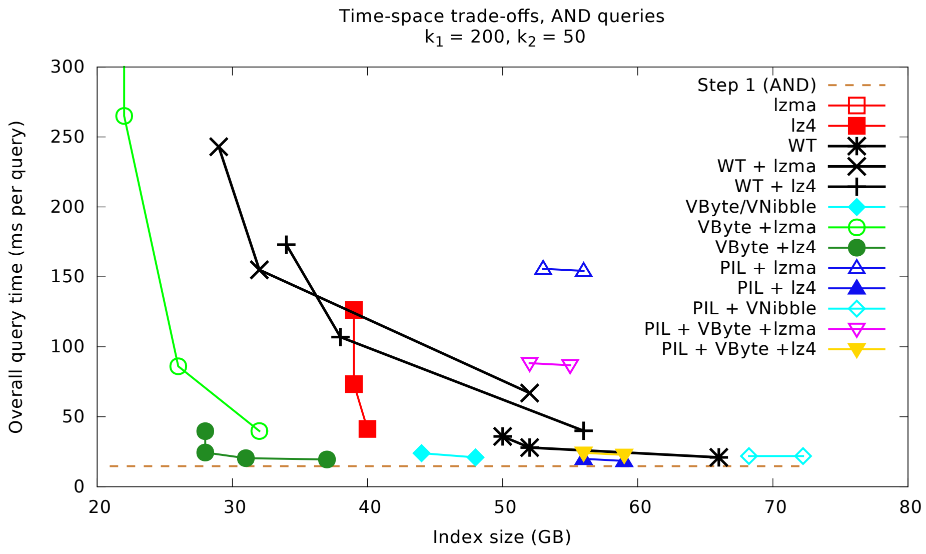


Figure A.3: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$ and $k_2 \in \{10, 50\}$, including Step 1(AND queries), Step 2 and Step 3. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

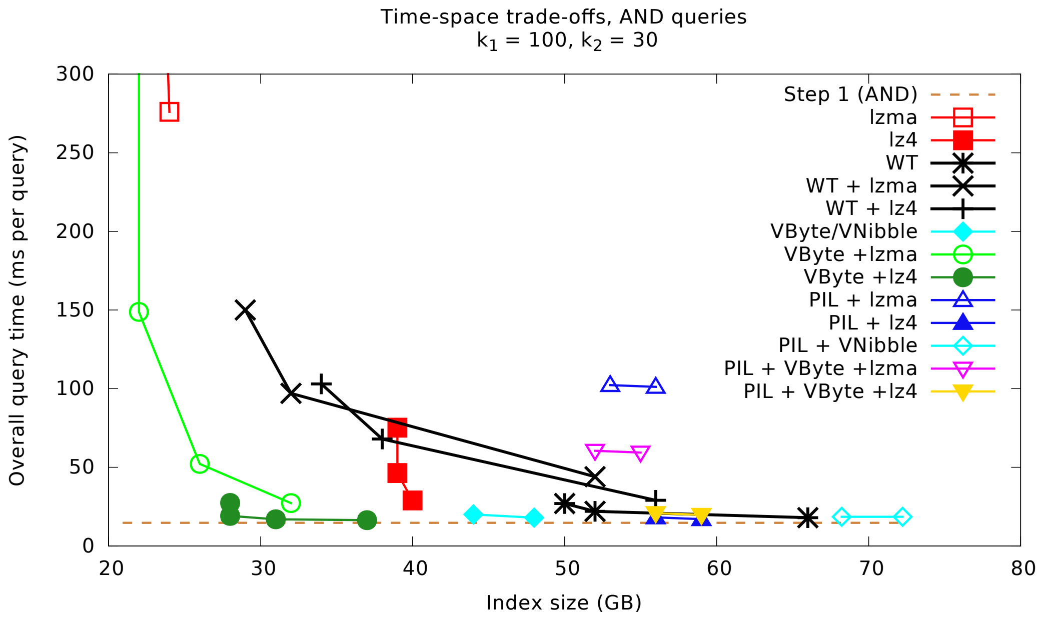


Figure A.5: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$ and $k_2 = 30$, including Step 1, Step 2 and Step 3.

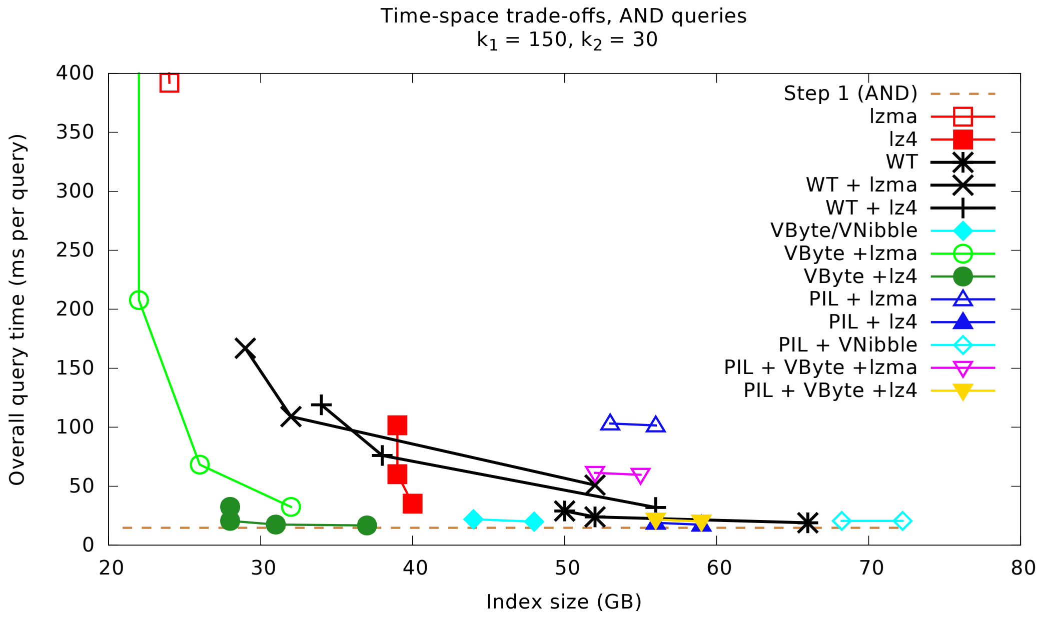


Figure A.6: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$ and $k_2 = 30$, including Step 1, Step 2 and Step 3.

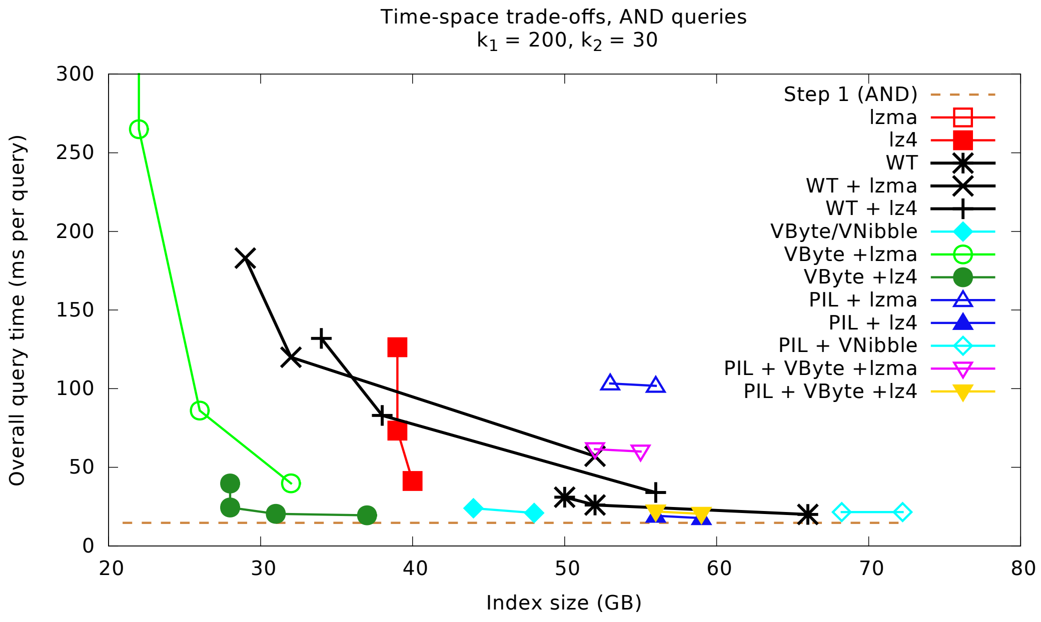


Figure A.7: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$ and $k_2 = 30$, including Step 1, Step 2 and Step 3. It is important to note that Scheme 1 has query time greater than 400 ms/q.

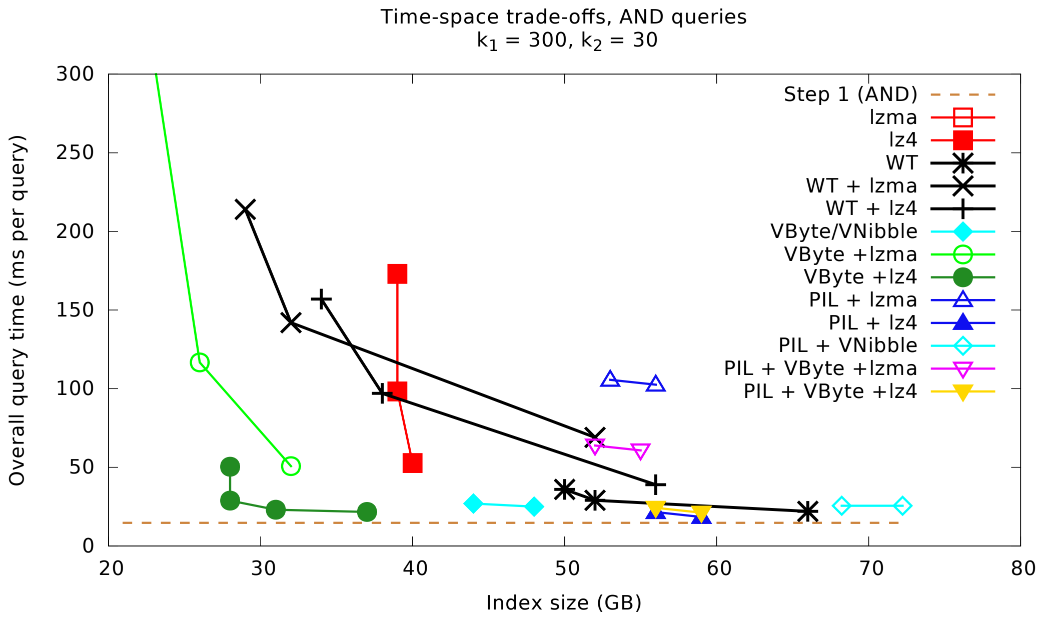


Figure A.8: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$ and $k_2 = 30$, including Step 1, Step 2 and Step 3. It is important to note that Scheme 1 has query time greater than 400 ms/q.

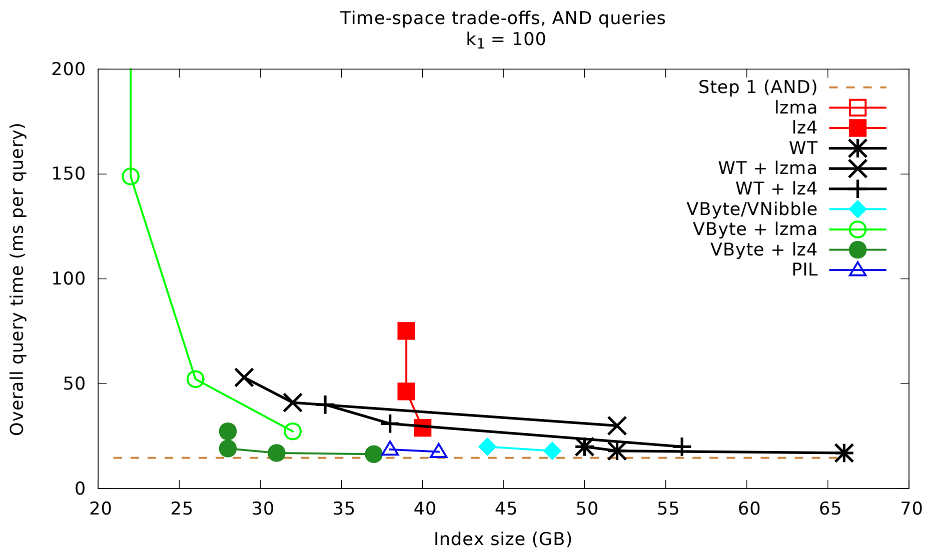


Figure A.9: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$, including Step 1(AND queries), Step 2. It is important to note that scheme `lzma` has query time greater than 250 ms/q.

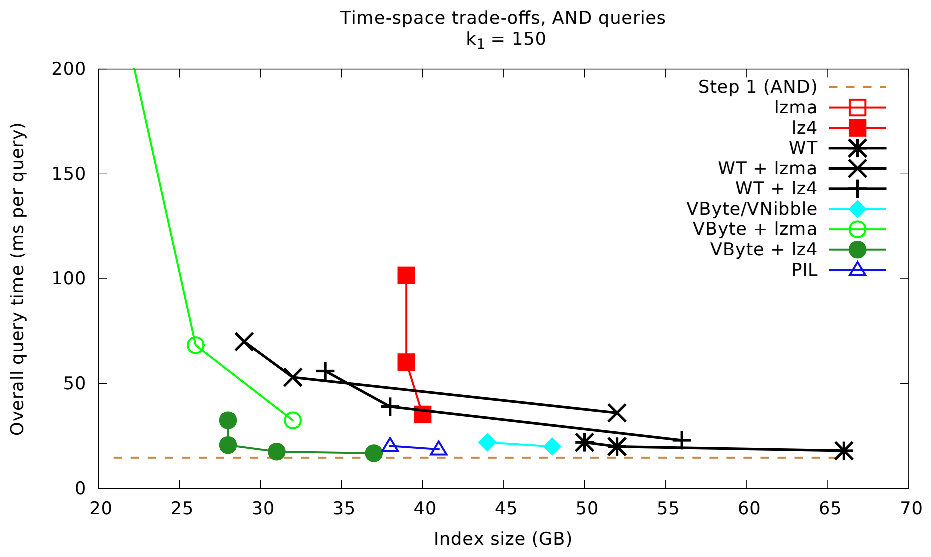


Figure A.10: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$, including Step 1(AND queries), Step 2. It is important to note that scheme `lzma` has query time greater than 250 ms/q.

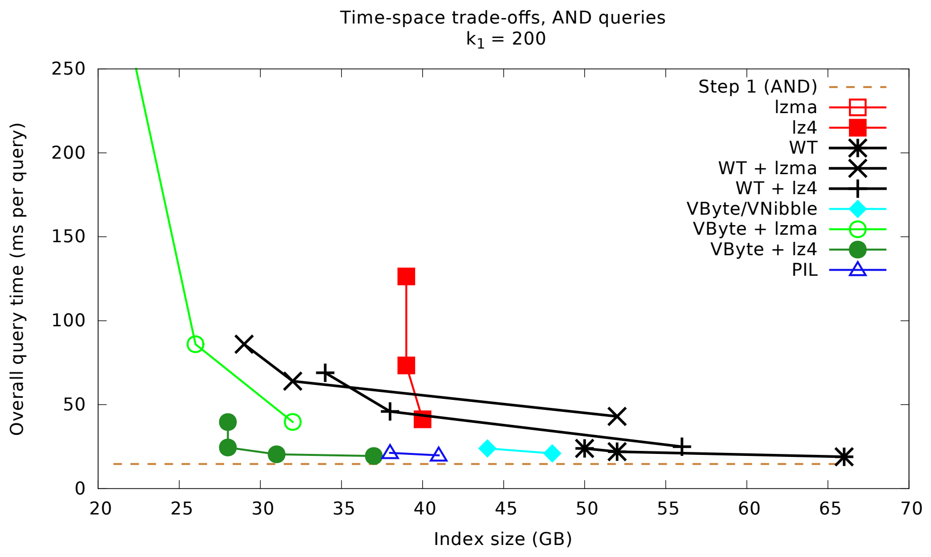


Figure A.11: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$, including Step 1(AND queries), Step 2. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

A.2 BMW OR Queries

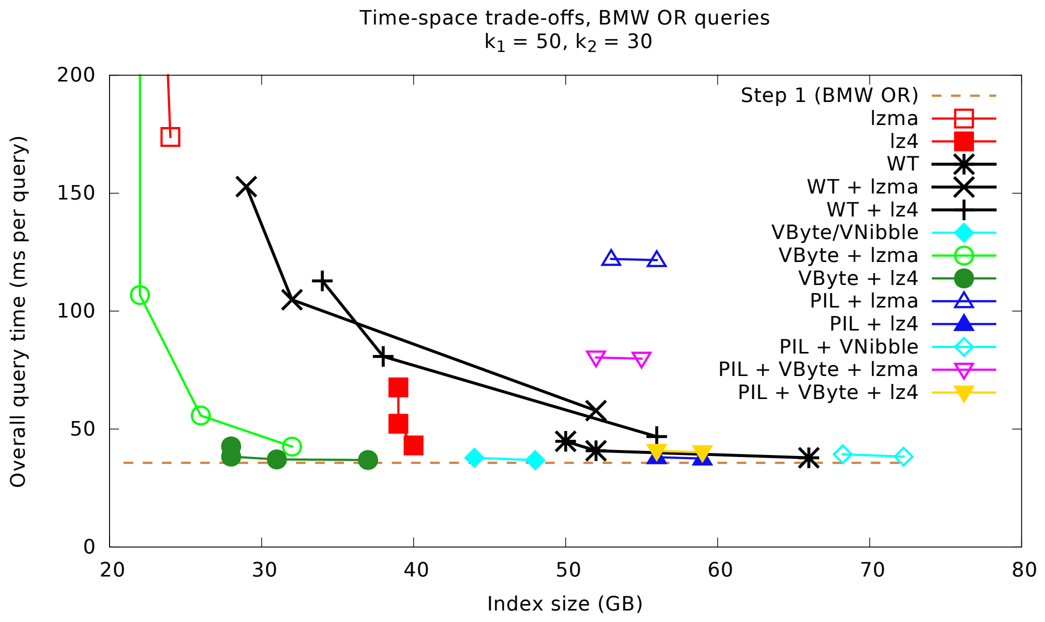
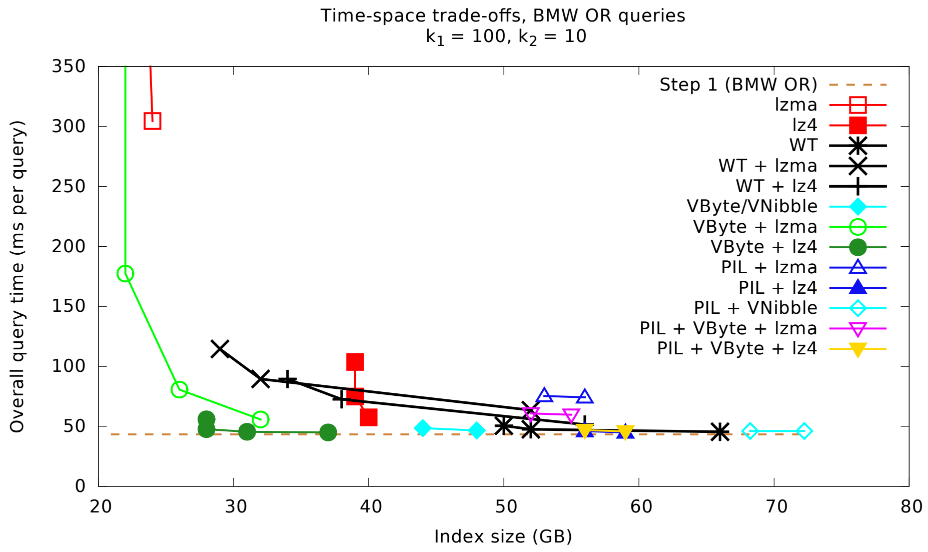


Figure A.15: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$ and $k_2 = 30$, including Step 1, Step 2 and Step 3.



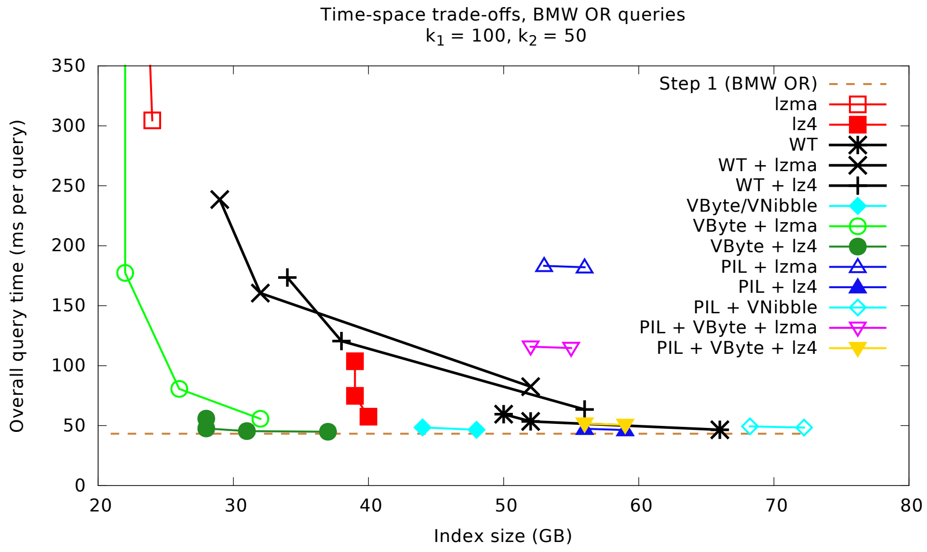
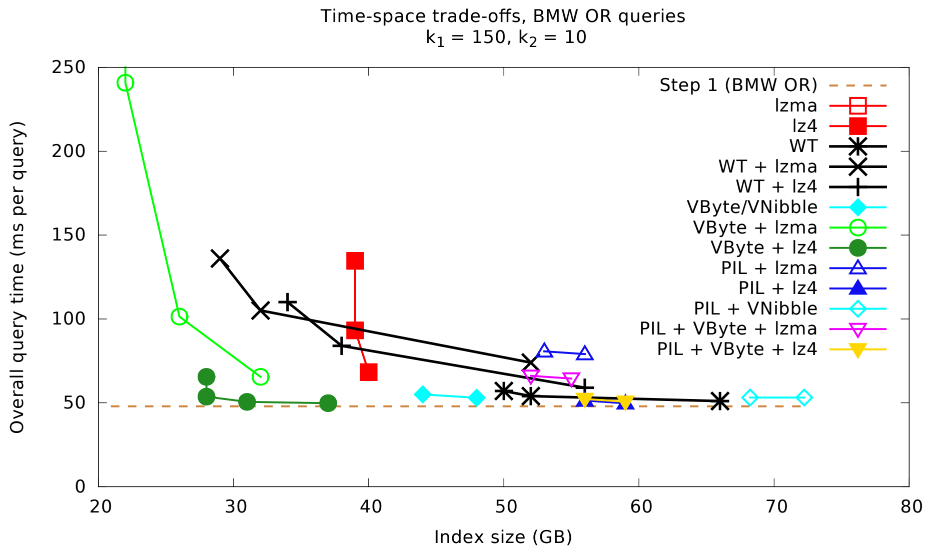


Figure A.12: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$ and $k_2 \in \{10, 50\}$, including Step 1(BMW OR queries), Step 2 and Step 3.



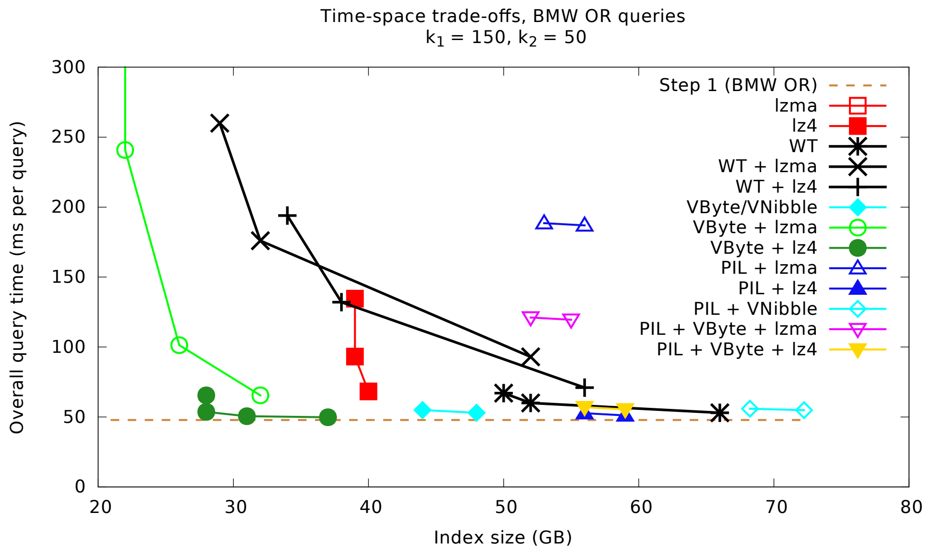
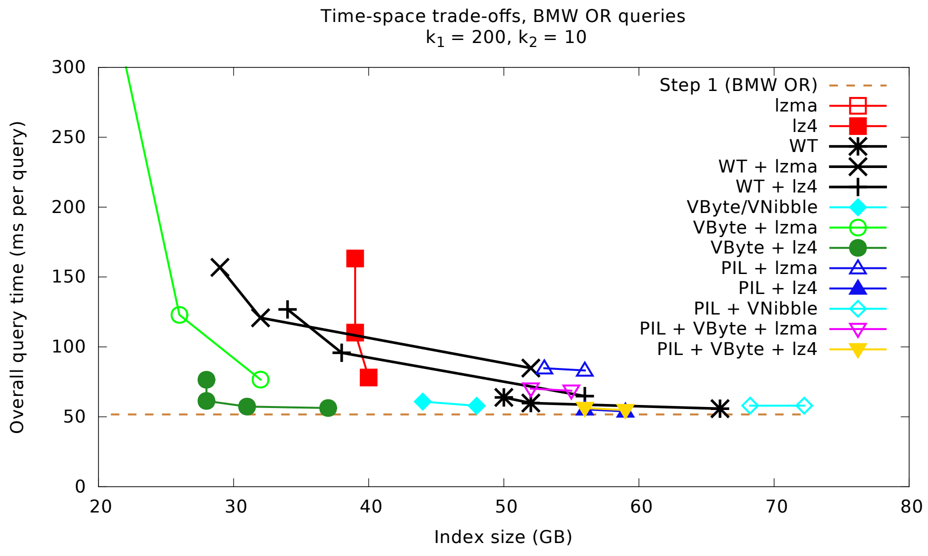


Figure A.13: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$ and $k_2 \in \{10, 50\}$, including Step 1(BMW OR queries), Step 2 and Step 3. It is important to note that scheme `lzma` has query time greater than 400 ms/q.



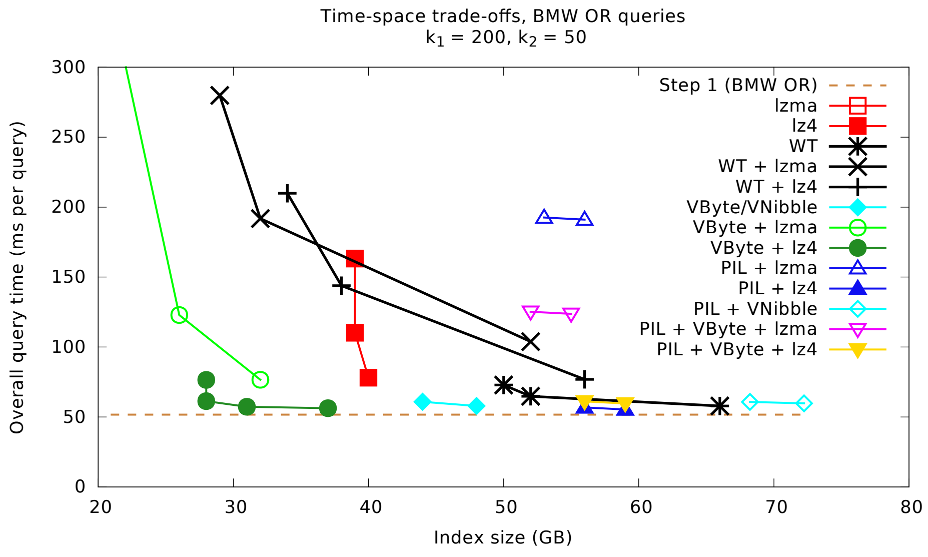


Figure A.14: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$ and $k_2 \in \{10, 50\}$, including Step 1(BMW OR queries), Step 2 and Step 3. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

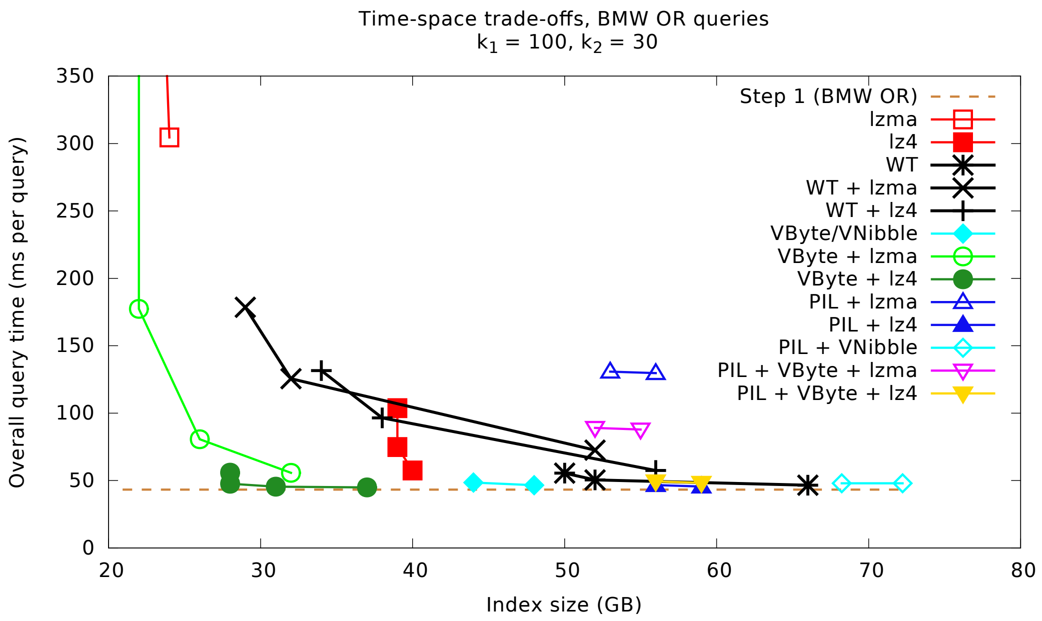


Figure A.16: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$ and $k_2 = 30$, including Step 1, Step 2 and Step 3.

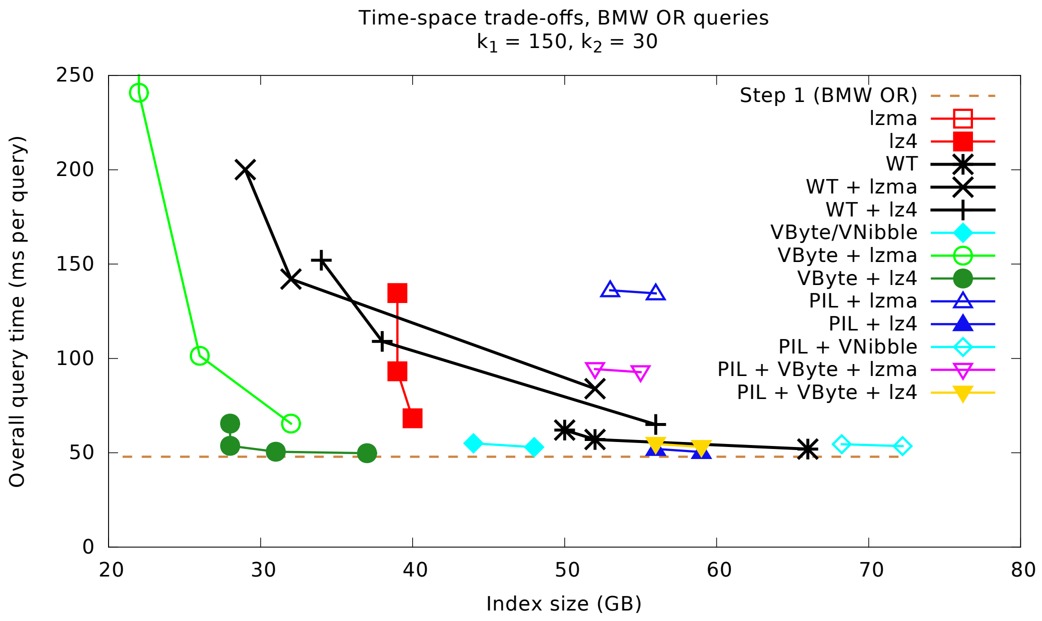


Figure A.17: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$ and $k_2 = 30$, including Step 1, Step 2 and Step 3. It is important to note that Scheme 1 has query time greater than 400 ms/q.

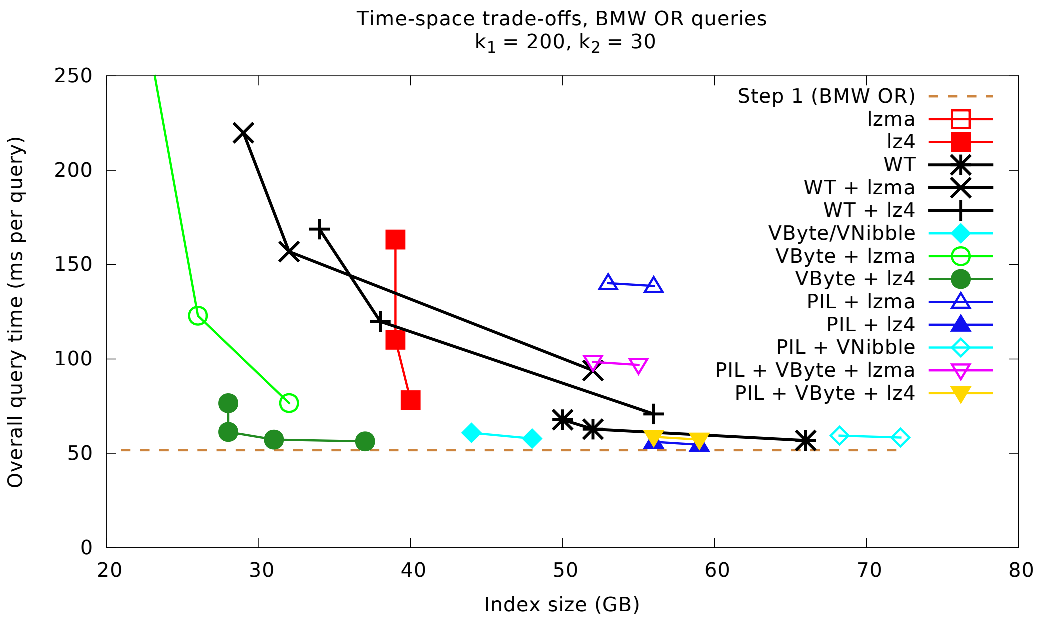


Figure A.18: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$ and $k_2 = 30$, including Step 1, Step 2 and Step 3. It is important to note that Scheme 1 has query time greater than 400 ms/q.

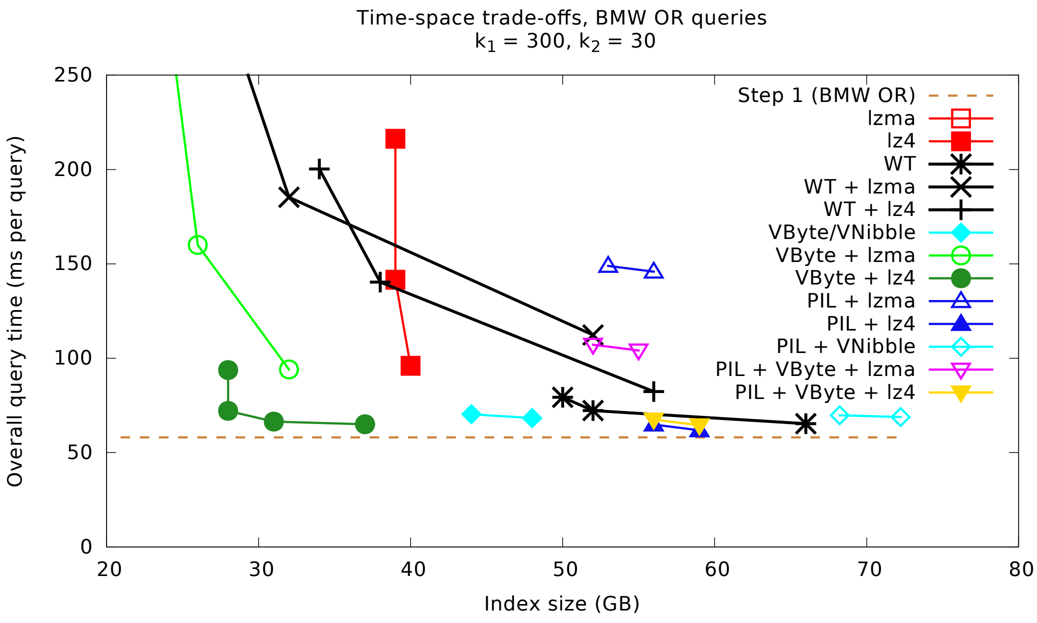


Figure A.19: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 300$ and $k_2 = 30$, including Step 1, Step 2 and Step 3. It is important to note that Scheme 1 has query time greater than 400 ms/q.

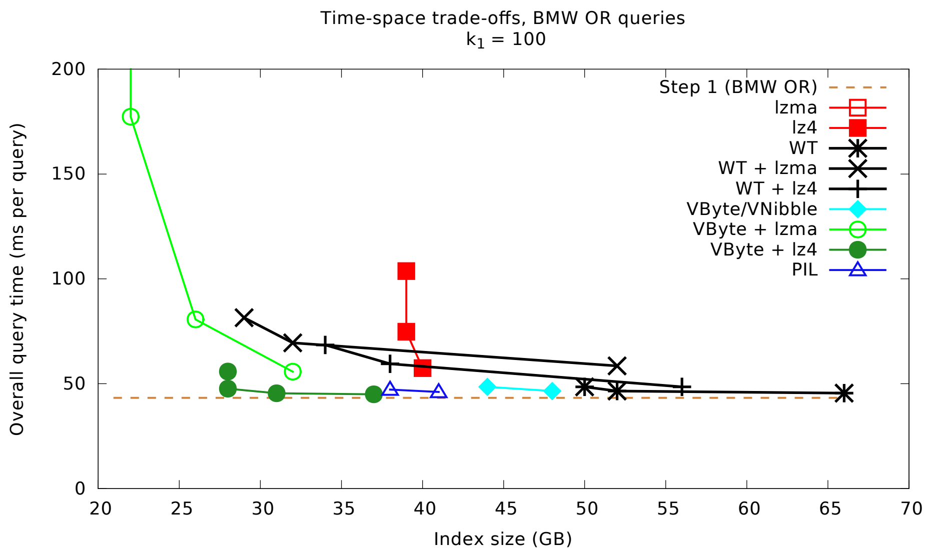


Figure A.20: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 100$, including Step 1(BMW OR queries), Step 2. It is important to note that scheme `lzma` has query time greater than 300 ms/q.

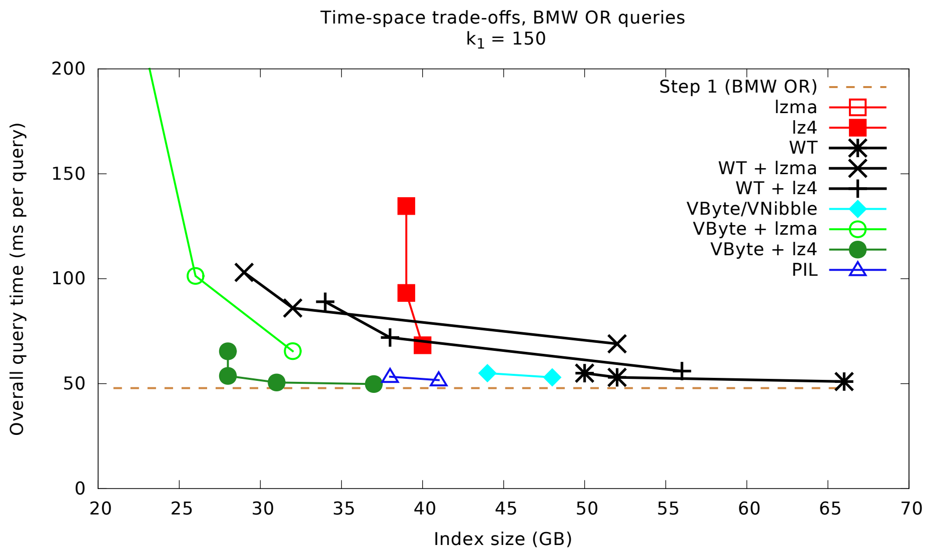


Figure A.21: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 150$, including Step 1(BMW OR queries), Step 2. It is important to note that scheme `lzma` has query time greater than 400 ms/q.

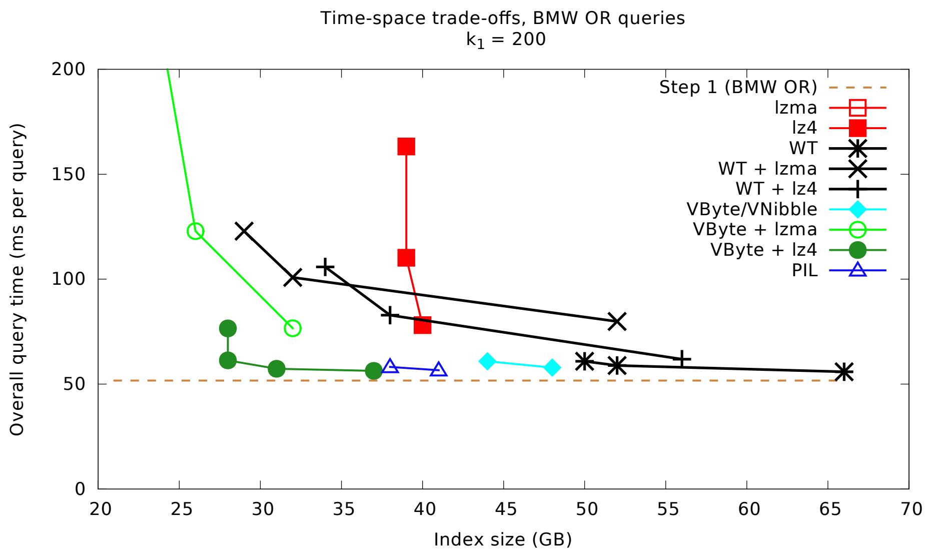


Figure A.22: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 200$, including Step 1(BMW OR queries), Step 2. It is important to note that scheme `lzma` has query time greater than 400 ms/q.