Document Identifier Reassignment and Run-Length-Compressed Inverted Indexes for Improved Search Performance

Diego Arroyuelo*
Dept. of Informatics,
Univ. Técnica F. Santa María.
Yahoo! Labs Santiago, Chile
darroyue@inf.utfsm.cl

Senén González
Department of Computer
Science, University of Chile.
Yahoo! Labs Santiago, Chile
sgonzale@dcc.uchile.cl

Mauricio Oyarzún
University of Santiago.
Yahoo! Labs Santiago, Chile
mauricio.silvaoy@usach.cl

Victor Sepulveda
Yahoo! Labs Santiago, Chile
vsepulve@dcc.uchile.cl

ABSTRACT

Text search engines are a fundamental tool nowadays. Their efficiency relies on a popular and simple data structure: the *inverted indexes*. Currently, inverted indexes can be represented very efficiently using index compression schemes. Recent investigations also study how an optimized document ordering can be used to assign document identifiers (docIDs) to the document database. This yields important improvements in index compression and query processing time. In this paper we follow this line of research, yet from a different perspective. We propose a docID reassignment method that allows one to focus on a given subset of inverted lists to improve their performance. We then use run-length encoding to compress these lists (as many consecutive 1s are generated). We show that by using this approach, not only the performance of the particular subset of inverted lists is improved, but also that of the whole inverted index. Our experimental results indicate a reduction of about 10% in the space usage of the whole index (just regarding docIDs), and up to 30% if we regard only the particular subset of list on which the docID reassignment was focused. Also, decompression speed is up to 1.22 times faster if the runs must be explicitly decompressed and up to 4.58 times faster if implicit decompression of runs is allowed. Finally, we also improve the Document-at-a-Time query processing time of AND queries (by up to 12%), WAND queries (by up to 23%) and full (non-ranked) OR queries (by up to 86%).

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.4 [**Systems**]: Textual databases

*Partially funded by Fondecyt Grant 11121556.

SIGIR'13, July 28–August 1, 2013, Dublin, Ireland.

General Terms

Algorithms, Experimentation, Performance

Keywords

Inverted index compression, document reordering, query processing.

1. INTRODUCTION

Inverted indexes are the *de facto* data structure to support the high-efficiency requirements of a text search engine [29, 4, 10, 18, 31]. This includes, for instance, providing fast response to thousands of queries per second and using as less space as possible, among others. Given a document collection $\mathcal{D}$ with vocabulary $\Sigma = \{w_1, \ldots, w_V\}$ of V different words (or terms), an inverted index for $\mathcal{D}$ stores a set of *inverted lists* $I_{w_1}[1..n_1], \ldots, I_{w_V}[1..n_V]$. Every list $I_{w_i}[1..n_i]$ stores a *posting* for each of the n_i documents that contain the term $w_i \in \Sigma$. Typically, a posting stores the document identifier (docID) of the document that contains the term, the number of occurrences of the term in this document (the term frequency) and, in some cases, the positions of the occurrences of the term within the document. The inverted index also stores a *vocabulary table*, which allows us to access the respective inverted lists.

The space required by the inverted lists is dominant in the space of the index. Therefore, lists are kept compressed, not only to reduce their space usage but also to reduce the transference time from disk—which can be up to 4–8 times slower if the disk-resident lists are not compressed [10, see Table 6.9–page 213]. To answer a query, the involved lists must be decompressed—fully or partially. Hence, fast decompression is a key issue to support quick answers.

Inverted index compression has been studied in depth in the literature [29, 18, 4, 10]. Usually, docIDs, frequencies and positions are stored separately, hence they can be compressed independently. Even though it is important to compress each of these components, this paper is devoted to compressing just the docIDs. Frequencies and term positions can be usually compressed using similar techniques to that used for docIDs. However, the improvements obtained in this paper will be tailored to docIDs. According to [30, 3],

docIDs correspond to about 65% of a docIDs+frequencies index. If we also consider positional information, docIDs correspond to about 20% of the overall space [3].

The docIDs of list I_{w_i} are compressed sorting them by increasing docID, to then represent the list using gap encoding: the first docID of the list is represented as it is, whereas the remaining docIDs are represented as the difference with the previous docID. We call DGap this difference. For instance, given the inverted list $\langle 10, 30, 65, 66, 67, 70, 98\rangle$, its DGap encoding is $\langle 10, 20, 35, 1, 1, 3, 28\rangle$. This generates a distribution with smaller numbers, in particular for long lists. As we shall see through this paper, many of these DGaps are actually 1s, which correspond to terms that appear in documents whose docIDs are consecutive. To support searching, inverted lists are logically divided into blocks of, say, 128 DGaps each. This allows us skipping blocks at search time, decompressing just the blocks that are relevant for a query, not necessarily the whole list. Among the existing compression schemes for inverted lists, we have classical encodings like Elias [13] and Golomb/Rice [14], as well as the more recent ones VByte [28], Simple 9 [1], Interpolative [19] and PForDelta [32] encodings. All these methods benefit from sequences of small integers.

The assignment of docIDs to a given document database is not trivial. This task—usually known as *document reordering*—consists in ordering the documents in $\mathcal{D}$, to then assign the docIDs following this order. For instance, a simple and effective method consists in ordering the documents according to their urls [23]. These are called *ordered document collections* and will be the focus of this paper. Document reordering is not always feasible, as discussed in [30]. However, there are many applications where this can be used. The advantage of assigning docIDs in an optimized way is that it yields smaller DGaps in the inverted lists, hence better compression can be achieved [5, 6, 21, 23, 24] and in general a much better inverted index performance [30, 26]. Reductions of up to 50% both in space usage and *Document at a Time* (DAAT) query processing have been reported [30, 26]. The main reason of this improvement in the query processing time is that the docIDs that are relevant to a query tend to be clustered within the same inverted lists blocks, hence less blocks need to be decompressed.

A remarkable feature of ordered document collections is that the number of 1 DGaps is increased. For instance, after reordering the TREC GOV2 document collection, the distribution of DGaps has almost 60% of 1s, whereas a random ordering yields just 11% of 1s. Actually, these 1s tend to form long runs in the inverted lists. Having runs of equal symbols in a sequence allows us to use run-length encoding [14]: we simply encode a run writing its length. There are, nowadays, five main ways to take advantage of these runs in the lists, namely:

1. To use compression approaches that have been particularly tuned to compress small DGaps [30].

2. To use compression approaches like *Interpolative Encoding* [19], which is suitable to compress small DGaps.

3. To use compression approaches like *Interval Encoding* [7], able to compress the runs in the lists.

4. To use compression approaches like the one proposed in [2] to compress long runs of 1s.

5. To use compression approaches like *VSEncoding* [25], which computes the optimal block partitioning to improve both compression ratio and decompression speed.

Approach (1) above compresses every 1 in a run *explicitly*. If one uses the OptPFD approach [30] and a given run of 1s is large enough, we can encode each 1 using one bit. Approach (2), on the other hand, is able to represent the runs *implicitly*, which is a desirable feature: when the integer encoding process gets into a run, it will encode the extreme values of the run and then each 1 within the run is not represented. That is, interpolative encoding deals with the runs in a natural way. The only detail is that, depending on where a run lies within the list, it could be split into several consecutive runs by the encoding process. This may be not optimal, yet efficient enough in most cases. The main drawback of approach (2) is that both the decoding and query-processing performance are not competitive [10, 30]. Approach (3) also encodes the intervals of consecutive docIDs (i.e., the runs of 1s in the DGap-encoded lists), storing the initial docID in the run, followed by the length of the run. All runs in a list are stored in a separate storage. The remaining docIDs (i.e., those that are not in any run) are called *residuals*. These are encoded using DGaps, as usual. Even though approach (3) is effective for web graph compression, it has two potential drawbacks in our information retrieval setting. First, the separate storage of the docIDs (i.e., intervals and residuals) is not adequate for DAAT query processing. Second, after removing the intervals from the list, bigger DGaps are generated, which is similar to the effect seen in [17]. Unfortunately, this worsens the compression ratio achieved. Approach (4) suggests that their method can compress runs of 1s in inverted lists. Yet, it is not studied in depth [2]. Finally, approach (5) could be used along with the techniques proposed in this paper to implicitly compress runs of 1s.

In this paper we study in depth how the inverted index compression and DAAT query processing are affected by the runs of 1s that are generated in ordered document collections. We conclude that properly managing these runs can lead to improvements in the space usage and query processing time. See Section 3 for a summary of our contributions.

2. PREVIOUS CONCEPTS

2.1 Inverted List Compression Schemes

We study here the basic compression schemes for inverted lists, which shall be used throughout this paper.

2.1.1 Variable-Byte Encoding

VByte encodes an integer using an integral number of bytes [28]. As the encoding is byte-aligned, it can be decoded faster. To obtain this encoding, the binary representation of the integer to be encoded is split into 7-bit chunks, adding an extra continuation bit (or flag) to every chunk. This flag indicates whether the current chunk is the last one in the representation of the integer or not. The compression ratio of VByte is not efficient when the inverted lists have small DGaps like 1, 2 or 3 (which are relatively frequent in large inverted lists, particularly for ordered collections).

2.1.2 Simple 9 Encoding

The *Simple 9* encoding (S9 for short) [1] aims at a fast decompression, yet achieving a competitive compression ratio. Assuming a machine word of 32 bits, we divide it into as

many chunks of equal size as we can, storing a DGap in each chunk. We use a 4-bit header to indicate the decoder how many DGaps have been stored in that word. Hence, we have 28 bits left to store the DGaps. There are 9 ways—hence its name—of dividing the remaining 28 bits into equal-size chunks: 1 chunk of 28 bits; 2 chunks of 14 bits; 3 chunks of 9 bits, 1 unused bit; 4 chunks of 7 bits; 5 chunks of 5 bits, 3 unused bits; 7 chunks of 4 bits; 9 chunks of 3 bits, 1 unused bit; 14 chunks of 2 bits; and 28 chunks of 1 bit.

To decompress an S9 word, its header is used to determine the case using a `switch` statement in C, where the 9 cases are hard-coded to improve decompression speed. In practice, the decompression speed is slightly better than VByte, and it has a better compression ratio in general.

2.1.3 PForDelta *Encoding*

The PForDelta encoding [32] divides an inverted list into *blocks* of, usually, 128 DGaps each. To encode the DGaps within a given block, it gets rid of a given percentage—usually 10%—of the largest DGaps within the block, and stores them in a separate memory space. These are the *exceptions* of the block. Next, the method finds the largest remaining DGap in the block, let us say x, and represents each DGap 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 indicates information about the compression used in the block, e.g., the value of b.

To retrieve the positions of the exceptions, we 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. This forms a linked list with these slots. In case that b is too small and cannot accommodate the offset to the next exception, the algorithm forces to add extra exceptions between two original exceptions. This increases the space usage when the lists contain many small numbers.

To decompress a block, we first take b from the header, and invoke a specialized function that obtains the b-bit DGaps. Each b has its own extracting function, so they can be hard-coded for high efficiency. Once we decode the DGaps, we traverse the list of exceptions of the block, storing the original DGaps in their corresponding positions. This step can be slower, yet it is carried out just for 10% of the block. 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 [30], achieving a high decompression speed.

2.2 Document Reordering

The *document reordering* problem (or, equivalently, the *document identifier assignment* problem) consists in assigning similar docIDs—ideally, consecutive docIDs—to documents that contain similar terms [5, 6, 21, 23, 24]. This generates smaller DGaps in the inverted lists. As discussed in [30], document reordering is not always feasible. In particular, when global measures of page ranking (e.g., PageRank [8] and Hits [16]) are used for early termination. However, some applications benefit from document reordering.

Previous work [30] studies the performance of different compression schemes for ordered document collections. One of the main conclusions from [30] is that some compression schemes are not suitable for these cases, e.g., PForDelta, since most DGaps are small numbers, like 1 and 2. Hence, small values of b should be used in each block (e.g., $b = 1$ or $b = 2$). However, recall that PForDelta forces to add extra exceptions when b is not enough to represent an offset, which makes it unsuitable in these cases.

Two new alternative are introduced in [30]. The first one is called New PForDelta (NewPFD from now on), which supports using any value of $b \geq 1$ without adding any extra exception. The trick is to use the b-bit slot to store just b bits of the offset, while the remaining bits of the offset are stored in a separate space of memory. At decompression time, the original offsets are reconstructed using bit masks. The second alternative, called Optimized PForDelta (OptPFD from now on), allows us to use a variable number of exceptions in each block. Hence, one can choose either to minimize the space usage or to maximize the decompression speed, yielding also different trade-offs. One of the main results from [30] is that, given a target decompression speed, NewPFD and OptPFD have a better space usage than PForDelta.

3. OUR CONTRIBUTIONS

In this paper we show that by properly representing the runs of 1s that appear in inverted lists when document reordering is used, relevant improvements in space usage and query time can be achieved. In particular, we introduce a method for focusing the generation of runs of 1s to a particular set of lists and then show how to compress these runs using methods that support efficient query processing. Overall, our results are:

1. We reduce the space usage of the most efficient alternatives in [30] by about 10%.
2. The decompression speed is up to 1.22 faster if the runs must be explicitly decompressed and up to 4.58 times faster if implicit decompression of runs is allowed.
3. We improve the DAAT query processing time of AND (by up to 12%), WAND (by up to 23%), and full OR queries (by up to 86%).
4. We show that by using our approaches, the number of docIDs and the list blocks that are decompressed at query time can be reduced by up to 55%.
5. We show that our approaches are more suitable to compress static caches of inverted lists.

4. DOCUMENT REORDERING METHODS TO GENERATE RUNS OF 1S

Inverted list compressors benefit from distributions with small DGaps. Document reordering (Section 2.2) yields small DGaps in the resulting inverted lists. E.g., in the inverted index of the TREC GOV2 collection under random document order, just 10.75% of DGaps have value 1. If, on the other hand, we sort the documents by urls [23], we obtain 60.30% of DGaps with value 1. Thus, the ordered case generates many 1 DGaps [12]. Actually, it can be observed that many of these 1s are grouped into long runs. Instead of regarding these 1s separately (as in previous work [30]) we deal with the runs. We aim at improving the space usage and

query processing time of current alternatives. Runs can be encoded more efficiently, e.g., using *run-length compression* [14]: a run of repeated symbols can be encoded indicating the symbol and then writing the length of the run. This approach has been successfully used in many areas [14, 20].

The existing document reordering methods are mainly based on the document collection to carry out the ordering. This yields smaller indexes with enhanced query time [30, 26] since, after reordering, the documents that are relevant to a query tend to group into a few inverted-list blocks. Thus, less blocks need to be decompressed at query time. However, typically in practice some inverted lists are more important than others. Hence, one would want to focus on improving their particular performance, as opposed to improving the whole index. A particular such application is that of static inverted-list caches: the aim is to maintain in main memory the most-frequently-queried inverted lists, as well as other big inverted lists that are eventually queried and whose big size makes prohibitively expensive to transfer them from secondary storage. This effect cannot be achieved if we use methods that first assign docIDs and then construct the index following this assignment.

We develop next a heuristic to assign docIDs such that it improves the compression of a particular set of inverted lists. Our method uses the inter-list dependencies (i.e., the list intersections) to assign docIDs. As a result, these intersections become runs when encoded as DGaps. This will improve the compression ratio of these lists, avoiding the space-usage problems of [17] in cases where document reordering is allowed. Also, in Section 6 we will show that the resulting run (i.e., the intersection) will be stored inside a single block, reducing the number of blocks that are decompressed at query time. In general, finding the docID assignment that optimizes the amount of runs in the lists is known to be an NP-hard problem [15]. Thus, our technique is a heuristic that attempts just to improve the distribution and quality of runs, though it may not be the optimal one.

4.1 A Flexible docID Assignment Method for Focusing the Creation of Runs

Let $\mathcal{I}$ be the inverted index for a given document collection $\mathcal{D}$, where docIDs have been assigned arbitrarily. Let $\mathcal{L} = \langle I_{i_1}, I_{i_2}, \ldots, I_{i_m} \rangle$ be an ordered subset of the inverted lists of $\mathcal{I}$, such that I_{i_j} is the jth list in $\mathcal{L}$. The document ordering process will be based on this ordered set of lists. Let $F : \mathbb{Z} \mapsto \mathbb{Z}$ be a function such that $(i, j) \in F$ iff docID i has been already reenumerated as docID j.

Initially, we set $F \leftarrow \emptyset$, and start the process from I_{i_1}, which stores docIDs $d_1, d_2, \ldots, d_l$. A simple approach could be to rename $d_1 \rightarrow 1$, $d_2 \rightarrow 2$, ..., $d_l \rightarrow l$. For each such d_i, we set $F \leftarrow F \cup \{(d_i, i)\}$. Notice how this transforms I_{i_1} into a single run when representing the list as DGaps. Now, we go on to reenumerate I_{i_2}, with the restriction that every docID i such that $(i, j) \in F$ (for a given j) cannot be reenumerated anymore during the process. These are called *fixed* docIDs. Hence, we assign consecutive docIDs (starting from $l+1$) to any docID i in I_{i_2} such that $(i, j) \notin F$, for any j, and add the corresponding pair to F. For instance, let us consider the following example: $I_{i_1} = \langle 10, 30, 65, 66, 67, 70, 98 \rangle$ and $I_{i_2} = \langle 20, 30, 66, 70, 99, 101 \rangle$. The first step reenumerates I_{i_1} assigning $10 \rightarrow 1$, $30 \rightarrow 2$, $65 \rightarrow 3$, $66 \rightarrow 4$, $67 \rightarrow 5$, $70 \rightarrow 6$, $98 \rightarrow 7$. Now we reenumerate I_{i_2}, having into account that docIDs 30, 66 and 70 are now fixed. Hence, we assign $20 \rightarrow 8$, $99 \rightarrow 9$, $101 \rightarrow 10$. After the process, the result is $I_{i_1} = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$ and $I_{i_2} = \langle 2, 4, 6, 8, 9, 10 \rangle$.

Notice from the previous example that after the first step, all docIDs in the intersection $I_{i_1} \cap I_{i_2}$ are fixed docIDs, hence they cannot be changed when processing I_{i_2}. Notice also how the inter-list dependencies complicate the run generation in I_{i_2}, as the documents in $I_{i_1} \cap I_{i_2}$ could have been reenumerated in any order when processing I_{i_1}.

We can fix this problem as follows. We instead start by reenumerating the docIDs in $I_{i_1} \cap I_{i_2}$, renaming them from 1 to $|I_{i_1} \cap I_{i_2}|$ and add them as fixed docIDs in F. Next, and since the remaining of both lists do not intersect each other, we can reenumerate them to generate a run in each list. In this way, one of the lists will become a single run when DGap encoded, whereas the other will contain two runs. For the previous example, we start reenumerating the intersection $I_{i_1} \cap I_{i_2}$ as $30 \rightarrow 1$, $66 \rightarrow 2$, $70 \rightarrow 3$. Then, we reenumerate the remaining elements in I_{i_1} as $10 \rightarrow 4$, $65 \rightarrow 5$, $67 \rightarrow 6$, $98 \rightarrow 7$. Finally, we reenumerate the remaining elements in I_{i_2} as $20 \rightarrow 8$, $99 \rightarrow 9$, $101 \rightarrow 10$. The resulting lists are $I_{i_1} = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$ and $I_{i_2} = \langle 1, 2, 3, 8, 9, 10 \rangle$. This generates more runs than the previous approach.

In general, we start from I_{i_1} and compute $I_{i_1} \cap I_{i_2}$. If $|I_{i_1} \cap I_{i_2}| \geq M$, for a given threshold M, we compute $I_{i_1} \cap I_{i_2} \cap I_{i_3}$, and repeat the process until $|I_{i_1} \cap \cdots \cap I_{i_{j+1}}| < M$. At this point, we take the d documents in $I_{i_1} \cap \cdots \cap I_{i_j}$ and assign them consecutive docIDs (e.g., from 1 to d). Next, we add them as fixed docIDs in F. This will generate a run of length d when computing the DGap encoding of I_{i_j}. Then, we go back and assign consecutive docIDs to the d' documents in $I_{i_1} \cap \cdots \cap I_{i_{j-1}}$. Notice that d of these docIDs are already fixed from the previous step, hence we only assign docIDs from $d+1$ to d' to the remaining ones. This generates a run of length d' in $I_{i_{j-1}}$. Eventually, this process gets back to I_{i_1}, where all documents in $I_{i_1} \cap I_{i_2}$ are fixed and will form a run. Next, the inverted lists $I_{i_1}, I_{i_2}, \ldots, I_{i_j}$ are removed from $\mathcal{L}$, and their non-yet-fixed docIDs are re-inserted into $\mathcal{L}$ as independent lists, following some well-defined order. This process is repeated until $\mathcal{L} = \emptyset$.

We call *intersection-based docID assignment* (IBDA) this process. It is important to note that the intersections:

- $I_{i_1} \cap I_{i_2}$ in either I_{i_1} and I_{i_2};
- $I_{i_1} \cap I_{i_2} \cap I_{i_3}$ in I_{i_3};
- ...;
- $I_{i_1} \cap \cdots \cap I_{i_j}$ in I_{i_j}

are transformed into respective runs in each of these lists after the first step of IBDA. As we shall learn through this paper, this gives us a way to precompute intersections as runs. We just need to be careful in setting a suitable initial order for the lists in $\mathcal{L}$: if, for some reason, we want to transform the intersection between two given inverted lists into a run, these lists must be consecutive in $\mathcal{L}$. This can have many advantages at query time (see Section 6).

To summarize, notice the flexibility of our method, because of the following:

- It allows any initial order in the lists to be processed.
- It is based on just an initial inverted index; no information about the underlying documents is needed.

- It allows any initial docID assignment.
- It allows any re-insertion order for the tails of the lists.

This allows us to adapt the method to different situations.

5. RUN-LENGTH COMPRESSION OF INVERTED LISTS

Instead of using Interpolative Encoding [19] or Interval Encoding [7] to deal with the runs, we adapt some of the most effective compression schemes to run-length encode runs of 1s in inverted lists. We aim at efficient query processing time and improved compression ratios.

5.1 Run-Length Encoded VByte

To run-length encode the runs of 1s with VByte, we need to set a special mark that indicates, at decompression time, that we are in the presence of a run. Since we are encoding DGaps > 0, we use the byte 00000000 as the special mark The idea is to replace $x \geq 3$ consecutive 1s in an inverted list by the 00000000 mark, followed by the VByte encoding of x. We call RLE VByte this scheme. RLE VByte is able to detect and handle very small runs (of length ≥ 3). We restrict the minimum length to be 3, because this yields the same space as for runs of minimum length 2 (in both cases the minimum run uses 2 bytes), yet the former has better decompression times. Indeed, this will be the only method in this paper able to deal with small runs of length $3 \leq \ell \leq 27$.

We introduce now an alternative implementation of the original VByte scheme, on which will be based the RLE VByte implementation. The idea is to take advantage, at decompression time, of several consecutive small DGaps. Assuming that 0 is used as terminator flag in VByte, we take 4 consecutive bytes from the encoding (i.e., a 32-bit word) and carry out a bitwise AND with 0X80808080. This allows one to determine at once the 4 terminator flags of these bytes. In particular, if the bitwise AND results in a 0, it means that the terminator flags are all 0, hence we have 4 DGaps encoded in 4 bytes. We can then hard-code all cases, which can make a difference if most DGaps use 1 byte.

A potential drawback of RLE VByte is that, at decompression time, we need to make an extra comparison per element in the list, to know whether it is the mark of a run or not. However, to decode a run, the original VByte has to make one comparison for every 1. RLE VByte, on the other hand, is able to decode the run more efficiently, as we shall see in our experiments.

5.2 Run-Length Encoded S9

S9 can be easily adapted to run-length encode the runs of 1s, as it is suggested for a variant of this scheme [2]. We only need to add a new S9 case, which will be used to encode runs—recall that we use just 9 out of 16 cases available for the 4-bit headers. The remaining bits of the S9 word are used to represent the length of the run. It is rather common in practice to find S9 words that encode 28 ones, followed by a different S9 case (indeed, this is one of the most frequent cases for GOV2 inverted lists).

We define RLE S9 as follows, in order to compress runs of 1s: **(C1)** 1 chunk of 28 bits; **(C2)** 2 chunks of 14 bits each; **(C3)** 3 chunks of 9 bits each, 1 unused bit; **(C4)** 4 chunks of 7 bits each; **(C5)** 7 chunks of 4 bits each; **(C6)** 9 chunks of 3 bits each, 1 unused bit; **(C7)** 14 chunks of 2 bits each; **(C8)** 28 chunks of 1 bit each followed by C1; **(C9)** 28 chunks of 1 bit each followed by C2; **(C10)** 28 chunks of 1 bit each followed by C3; **(C11)** 28 chunks of 1 bit each followed by C4; **(C12)** 28 chunks of 1 bit each followed by C5; **(C13)** 28 chunks of 1 bit each followed by C6; **(C14)** 28 chunks of 1 bit each followed by C7; **(C15)** 28 chunks of 1 bit each followed by 5 chunks of 5 bits each; **(C16)** there are two cases still missing. The first one consists of a word storing 5 chunks of 5 bits each. The second case corresponds to runs of 1s longer than 28. Since the case of 5 chunks of 5 bits each has 3 bits unused, we use one of them to make the difference between these cases. If the bit is a 0, then we have 5 chunks of 5 bits each. If the bit is 1, then we use the remaining 27 bits to represent the length of the run.

Cases 1 to 15 are identified with 4-bit headers from 0000 to 1110. The remaining two cases use 5-bit headers 11110 and 11111. These variable-length headers can be uniquely decoded very efficiently using a `switch` statement in C, checking just 4-bit headers as usual. If the 4-bit header is 1111, we check the following bit to determine the 5-bit header.

The compression process is carried out in two steps: (*1*) We first compress the list using the original S9 algorithm; (*2*) we traverse the S9 words generated in the previous step, and replace the S9 words that store 28 ones by one of the new cases defined above: if there are multiple consecutive such S9 words, we replace them by case 11111; if instead we have one such word followed by a different case, we replace it by the corresponding case 8 to 15.

5.3 Run-Length Encoded PForDelta

We define now RLE PFD, which adds run-length-encoding capabilities to PForDelta, with minor changes to the original scheme. We define two kind of blocks: (*1*) *normal blocks*, containing 128 DGaps (as usual), and prefixed by a 32-bit block header; (*2*) *run blocks*, which encode a run storing its length within the 32-bit header. We need an extra bit in the header to indicate whether it corresponds to a normal block or encodes a run length. Fortunately, the original PForDelta leaves a unused bit in the header. At decompression time, the flag is checked to see whether one must decompress a normal block or a run.

This scheme is unable to encode relatively short runs. First, we use 32 bits (the header) to represent the run length. Hence, we define run blocks as encoding runs of length ≥ 32 (hence, we use at most 1 bit to encode each 1 in the run). Second, it is very likely that some of the 1s lying at the beginning of a run will be used to fill the preceding normal block. Just the remaining 1s could be used to form a run—provided they are 32 or more. Hence, runs of moderate length ($\approx$ 100–200)—which are rather frequent in practice—are easy to detect. We can have, for instance, that the first docID in an inverted list is $i \neq 1$, and next it has a run of 158 1-DGaps. RLE PFD will not be able to encode the run as such: the first block will be a normal block storing i and then 127 1-DGaps. The next will be also a normal block storing the 31 remaining 1-DGaps (which cannot be regarded as a run, as we already said).

5.4 Experimental Evaluation

For our experiments we use 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 L2 cache, and a 96GB RAM, running version 2.6.34.8-68.fc13.i686.PAE of Linux kernel.

We index the 25.2 million documents from the TREC GOV2 collection. We implement our RLE schemes using C++. We compile our codes with `g++ 4.4.5`, with optimization flag `-O5`. For PForDelta, NewPFD, and OptPFD we use the highly-efficient implementations from [30]. We base our implementation of RLE PFD on OptPFD. VByte and S9 implementations are from ourselves. For VByte, S9, PForDelta, NewPFD and OptPFD, we subtract 1 to each DGap in the index, so they start from zero.

5.4.1 Document Reordering Methods

In our experiments, we use the following document orders to assign the docIDs to the TREC GOV2 collection: (1) the original order given to the documents in the collection ("Unsorted" in our tables); (2) the URL sorting ("URL" in our tables); (3) the IBDA ("IBDA" in our tables) method introduced in Section 4.1.

In our experiments, we found out that IBDA works better on inverted indexes for already-sorted document collections. Thus, we apply IBDA to the inverted index constructed for the URL-sorted document collection. In our particular implementation of IBDA, we sort the inverted lists of the index according to the following procedure (recall from Section 4.1 that we need an ordered set of list $\mathcal{L}$). We take the 10,000 first queries from the TREC 2006 query log and compute the pairs of terms that are most frequently queried. We then sort the inverted lists such that if the most-frequently-queried pair of terms is w_i and w_j, hence the inverted lists I_{w_i} and I_{w_j} are the first lists in $\mathcal{L}$. We then take the second most-frequent pair of terms and add their inverted lists to $\mathcal{L}$ (just in case they have not being added before). This procedure is repeated until all frequent pairs have been added. Notice how in this way we force the inverted lists of the most-frequently-queried pairs of terms to be in consecutive positions of $\mathcal{L}$. The result is that the intersection of these pairs is transformed into a run by IBDA. This precomputes these intersections to speed-up query processing. All lists that do not appear among the most frequent pairs are sorted by length, from longest to shortest. In each step of the algorithm, the non-processed list tails are re-inserted according to their lengths.

5.4.2 Space Usage

Table 1 shows the experimental space usage of the docID data of the whole inverted index, for different compression schemes and the various docID assignment schemes we tested.

The main conclusions are the following:

- The space usage of either interpolative encoding, VByte, S9 and OptPFD is slightly improved when using our IBDA enumeration method, rather than just sorting by URL.
- When we use the RLE-encoded schemes on docIDs assigned by URL and IBDA, the space usage improves considerably compared with the corresponding normal schemes. The best improvement is achieved by RLE VByte: a reduction in space usage of 44.58%. This is because relatively-short runs are rather frequent in inverted indexes and, unlike other schemes, RLE VByte is able to encode runs of length ≥ 3.

Table 1: Overall space usage (in MB) of the TREC GOV2 inverted index, for different compression schemes and docID assignment methods. The space includes just the docIDs.

Compression Scheme	Reorder Method		
	Unsorted	URL	IBDA
Interpolative Encoding	5,507	2,712	2,641
VByte	7,526	6,726	6,754
S9	6,687	3,777	3,735
OptPFD	6,348	4,600	4,504
RLE VByte	7,418	3,861	3,743
RLE S9	6,687	3,455	3,392
RLE PFD	6,384	4,264	4,137

- Disregarding interpolative encoding, the smallest space usage is achieved by RLE S9 on docIDs assigned by IBDA. The improvement over S9 on docIDs assigned by URL (the previously-known best performer [30]) is of about 10.19%.
- Finally, RLE S9 uses between 25.82% and 28.44% extra space on interpolative encoding, depending on the docID assignment scheme used.

5.4.3 Decompression Speed

To test decompression speed, we use the TREC 2006 query log and decompress the inverted lists corresponding to the query terms. Table 2 shows the average decompression speed (in millions of DGaps per second) for the different compression schemes and docID assignment methods we have studied. All RLE methods *implicitly* decompress the runs.

Table 2: Average decompression speed (in millions of DGaps per sec.) for different compression schemes and docID assignment methods, on the TREC GOV2 inverted index.

Compression Scheme	Reorder Method		
	Unsorted	URL	IBDA
Interpolative Encoding	43.94	43.05	64.44
VByte	818.81	827.98	917.47
S9	749.28	983.84	1199.61
OptPFD	927.02	857.48	852.65
RLE VByte	446.58	1,313.03	1,988.86
RLE S9	779.52	1,812.22	2,691.09
RLE PFD	989.74	2,026.54	3,931.73

The main conclusions are the following:

- Except for OptPFD, using IBDA (instead of just URL) increases the decompression speed of either interpolative (49.67%), S9 (21.93%) and VByte (10.81%).
- The decompression performance of our VByte variant (recall Section 5.1) is faster than the original implementation of VByte (which in our tests decompresses 571 million docIDs per second for Unsorted, 576 millions for URL, and 594 millions for IBDA). This is because 1-byte DGaps are the most frequent in inverted indexes and our implementation takes advantage of

this fact. Notice that the decompression speed of our VByte implementation is competitive with S9. Dean [11] shows similar approaches to group the flag bits to decode them fast. We think that our implementation can be also useful to speed-up the decompression of text snippets [27].

- RLE S9 on URL yields an enhancement of 84.19% over S9 on URL. For RLE PFD on URL the improvement is of 136.34%. RLE S9 on IBDA, on the other hand, increases the decompression speed by 173.53% over S9 on URL. Finally, for RLE PFD on IBDA the decompression speed is improved by 358.52%.
- RLE PFD shows an enhancement of decompression speed of 34.22% compared to PForDelta on URL. If, on the other hand, we consider RLE PFD on IBDA rather than PForDelta on IBDA, the improvement is of 45.32%.

The results in Table 1 and Table 2 indicate that RLE S9 on IBDA offers the best space vs. decompression-time tradeoff. We can conclude that IBDA by itself is able to improve decompression speed by up to 22% on URL. However, using the RLE encodings on IBDA data is the key to obtain the remarkable speedups that we mention above.

6. RUNS OF 1S AND DAAT PROCESSING

Document-at-a-Time (DAAT) and *Term-at-a-Time* (TAAT) are the usual ways to support efficient query processing in inverted indexes [10]. DAAT is generally faster and uses less memory than TAAT. Next we adapt DAAT query processing to efficiently handle the runs of 1s in the lists.

6.1 DAAT Query Processing

DAAT processes simultaneously—and from left to right—the inverted lists $I_{w_{i_1}}, \ldots, I_{w_{i_q}}$ of the query terms. This process is carried out with function `nextGEQ`(I_{w_j}, d), which yields the smallest docID d' in the inverted list I_{w_j} such that $d \leq d'$. Since `nextGEQ` will be invoked with increasing values of d, every inverted list maintains a *cursor* `curr` with the current position in that list. Thus, function `nextGEQ` moves the cursor forward and each invocation to `nextGEQ` starts from the position where the previous invocation stopped.

To support DAAT, inverted lists are compressed using a block layout: each list is divided into blocks of a given amount of DGaps—typically, 128 DGaps per block. For each block, a *block header* is maintained. Each such header stores, among other things, the largest absolute docID of the block. This is used for skipping at query time. Assume that we must search for docID d in inverted list I_{w_j}. Hence, we invoke `nextGEQ`(I_{w_j}, d). This function starts from position `curr` and moves through the block headers comparing d with the last docID in each block, skipping all non-relevant blocks. Once we arrive at the block that potentially stores d, we fully decompress it into an auxiliary `Buffer`. (This is not completely true if the block has been compressed using VByte, where a DGap per DGap decompression and checking is more efficient.) Next, we move `curr` through `Buffer` from left to right, looking for d.

6.2 DAAT and Implicit Run Decompression

As in Section 5.4.3, we do implicit decompression of runs, this time to support DAAT query processing. That is, if when decompressing a block we detect that a run has been compressed, we do not write every such 1 in `Buffer`. Instead, we just write the special mark `0` in the corresponding position of `Buffer`, followed by the length of the run. In other words, the time needed to decompress the whole run will be that needed to decompress just its length (i.e., a single integer) from our RLE schemes.

Assume that, at query time, we invoke `nextGEQ`(I_{w_j}, d) and that the block that potentially stores d has been decompressed into `Buffer`. Suppose that, looking for d, we reach position j of `Buffer` and it holds that `Buffer`$[j] = 0$. That is, we have reached a run of 1s of length $\ell =$ `Buffer`$[j+1]$. Let d' denote the docID stored at position $j-1$ in `Buffer`. Notice that the docIDs represented in the run lie in the interval $[d'+1, d'+\ell]$. If it holds that $d'+\ell < d$, we can safely skip the run and go on from position $k+2$ in `Buffer`. Otherwise, if $d'+\ell \geq d$ holds, the sought docID lies within the run, hence we cannot skip it. Notice that $p = d - d'$ is the position within the run corresponding to docID d. In this case we set $d' = d$ (indicating to the subsequent `nextGEQ` invocation that we have processed up to docID d) and return to the caller. We maintain the values of p and d' for the next search on the list. Assume now that we invoke `nextGEQ`(I_{w_j}, d''), for $d'' > d$. Assuming that d'' lies within the same block as d, we check whether $d'+\ell-p < d''$ holds. If that is the case, we can skip the run and go on form position $k+2$ in `Buffer`. Otherwise, d'' also lies within the run, hence we act as before, updating d' and p.

This approach can be seen as an abstract optimization of inverted lists, since we only need to change the way in which function `nextGEQ` is implemented. Hence, this approach can in principle be used by any DAAT-like algorithm [9, 22], obtaining the gains in space usage of Table 1.

6.3 Redefining the Block Layout of the Lists

We need to redefine the block structure of the lists, since they store long segments that have been implicitly represented—the runs. If, as usual, we force the blocks to store a fixed number of DGaps, then we would need to break some long runs in order to accommodate them within a block. This would dilute the benefits of having long runs. Lists are divided into blocks of fixed size because whole blocks must be decompressed to search inside then (as we saw before). Hence, a block cannot store too many DGaps. However, we do not need to decompress runs explicitly, thus the cost of decompressing a run is actually the same as decompressing a single integer. Also, handling a run in `Buffer` does not incur in considerable extra costs, just checking whether we can skip the run or not. Thus, we define blocks of fixed size, say 128 DGaps, and regard every run as if it were a single DGap. For RLE S9, if it happens that the 128th DGap lies in the middle of an S9 word, then we include the whole S9 word within the block. Hence, S9 blocks can have slightly more than 128 DGaps. In the case of RLE PFD, as we already saw in Section 5.3, it handles runs of length ≥ 32.

Notice that in this way the number of blocks in the lists could be reduced, except for RLE PFD. As a result, we would need to store less block headers, thus reducing the space usage. Also, we will see in the experiments below that the number of blocks decompressed at query time is reduced when compared to the results in [30], improving the decompression effectiveness.

6.4 Experimental Evaluation

We test now the efficiency of our approaches at query time. We use the original TREC 2006 query log for the GOV2 collection.

6.4.1 Space Usage

Table 3 shows the space usage of the TREC GOV2 inverted index. This time we include the space required by the block headers. Disregarding interpolative encoding, the smallest space is achieved by RLE S9 on docIDs assigned by IBDA. The space is about 11.08% smaller than S9 on URL and uses 21.70% extra space on that of interpolative encoding.

Table 3: Space usage (in MB) of the TREC GOV2 inverted index, for different compression schemes and docID assignment methods. The space includes the docIDs and the block headers.

Compression Scheme	Reorder Method		
	Unsorted	URL	IBDA
Interpolative Encoding	6,551	3,756	3,685
VByte	8,264	7,464	7,497
S9	7,524	4,601	4,572
OptPFD	6,835	5,087	5,105
RLE VByte	8,448	4,626	4,485
RLE S9	7,524	4,147	4,091
RLE PFD	7,267	5,003	4,850

6.4.2 AND Queries

Table 4 shows the experimental query time (in milliseconds per query) for different query processing algorithms (in particular, AND, WAND and OR). We compare different compression schemes and reordering methods.

For AND queries, S9 on URL sorting (the best existing trade-off from [30]) achieves 5.53 msec/q (according to our experiments). Using our IBDA technique to order the document collection and then our RLE PFD approach to compress the resulting docID inverted lists, we obtain 4.84 msec/q (an improvement of 12.11%). If, alternatively, we use RLE S9 on IBDA, we obtain 5.14 msec/q (an improvement of 7.05% over S9 on URL). Notice also the following: if we consider S9 and RLE S9, both on URL assignment, the query times changes just slightly: from 5.53 to 5.52, respectively. If, on the other hand, we consider using IBDA instead of URL to assign docIDs, the reduction of query time is bigger. This shows again the effectiveness of IBDA, as it is able to improve the already highly-efficient query times of AND queries [30] (recall, from Section 5.4.1, that we have used a query log to carry out IBDA).

6.4.3 WAND Queries

We also test with the WAND query-processing algorithm [9], which is rather popular in current search engines. We use tf-idf ranking and look for top-10 results. As it can be seen in Table 4, we are able to speed-up the query processing by using IBDA and the RLE compression methods. For instance, by using RLE S9 on IBDA docIDs we can enhance the query time from about 60 msec/q to about 47 msec/q (an improvement of about 20.84%). For RLE PFD on IBDA, the improvement is of about 23.44%. Table 5 shows the average

Table 4: Experimental query time for the different query processing algorithms we tested.

Query Algorithm	Compression Scheme	Reorder Method		
		Unsorted	URL	IBDA
AND	VByte	42.85	22.88	28.88
	S9	11.97	5.53	5.43
	OptPFD	11.90	5.57	5.56
	RLE VByte	19.73	7.49	6.91
	RLE S9	12.67	5.52	5.14
	RLE PFD	12.46	5.34	4.84
WAND (top-10)	VByte	175.71	76.93	75.27
	S9	145.99	59.50	50.34
	OptPFD	152.03	60.73	52.87
	RLE VByte	184.74	66.95	54.57
	RLE S9	150.87	59.14	47.10
	RLE PFD	153.14	58.77	46.49
OR	VByte	330.39	292.07	288.38
	S9	380.90	331.88	327.30
	OptPFD	357.28	323.90	319.15
	RLE VByte	340.29	83.99	41.62
	RLE S9	475.70	141.58	59.97
	RLE PFD	465.35	177.10	66.03

number of docIDs (in millions) that are decoded per query (runs are regarded as single integers) and average number of blocks (in thousands) that are decompressed per query. Our

Table 5: Millions of docIDs decoded on average per query ("Ints.") and thousands of blocks decompressed on average per query ("Blks.") for WAND. The processing is carried out for top-10 results.

Comp. Scheme	Reorder Method					
	Unsorted		URL		IBDA	
	Ints.	Blks.	Ints.	Blks.	Ints.	Blks.
VByte	10.07	63.29	5.34	24.04	6.23	18.24
S9	10.98	82.42	4.70	34.53	4.50	33.00
OptPFD	10.91	85.30	4.62	36.11	4.42	34.51
RLE VByte	6.93	60.53	1.71	15.51	1.30	11.58
RLE S9	10.89	81.85	3.21	24.21	2.09	15.87
RLE PFD	10.91	85.44	3.60	31.90	2.22	18.24

results indicate that the number of docIDs decoded (as well as the number of decompressed blocks) is reduced in two ways: first, by using an RLE compression scheme; second, by using IBDA rather than URL sorting. For instance, if we use RLE S9 on URL sorting instead of S9 on URL, we obtain a reduction of about 30% in decoded docIDs (the same figure can be observed for decompressed blocks). If we now use RLE S9 on IBDA sorting, we obtain a further reduction of about 34% over RLE S9 on URL. This shows that we are able to improve the decompression effectiveness, as we can process the same queries decompressing less blocks and docIDs. This is not only effective to reduce the query time (as we can see in Table 4) but also in cases where accessing blocks is expensive (for instance, when blocks need to be transferred from secondary storage).

Finally, we think that this query-time reduction can be also achieved by the Block-Max WAND algorithm from [22].

6.4.4 *Full-OR Queries*

We use now the run-length encoding of inverted lists to improve the performance of OR queries. In particular, full OR queries, where the full result (rather than a ranking) must be obtained. The main application is the offline merging of inverted lists.

Let I_1 and I_2 be the involved inverted lists. The search algorithm proceeds as usual for DAAT OR queries. However, if while processing I_1 we arrive at a run whose docIDs define the interval $[d', d' + \ell']$, we switch to I_2 and look for $d' + \ell' + 1$ in it. If we find that $d' + \ell' + 1$ also lies within a run $[d'', d'' + \ell'' + 1]$ in I_2, then we switch to I_1 again and look for $d'' + \ell'' + 1$ in it. We keep repeating this process until the current docID d^* we are looking for is not within a run. In this case, we can report the run-length encoding of the interval $[d', d^* - 1]$. This allows us to compute the union of these intervals without decompressing them. Notice also that runs are written in the output already encoded. We call RLE OR this process.

Table 4 shows the experimental results. For VByte, S9 and PForDelta we use the traditional DAAT OR processing. As it can be seen, RLE OR introduces relevant improvements. In particular, for RLE VByte on IBDA, where we obtain a reduction of 85.75% compared to VByte on URL. The rationale behind this exceptional performance of RLE VByte is that it is the only RLE scheme able to catch short runs. These are rather frequent in practice, which explains the good performance.

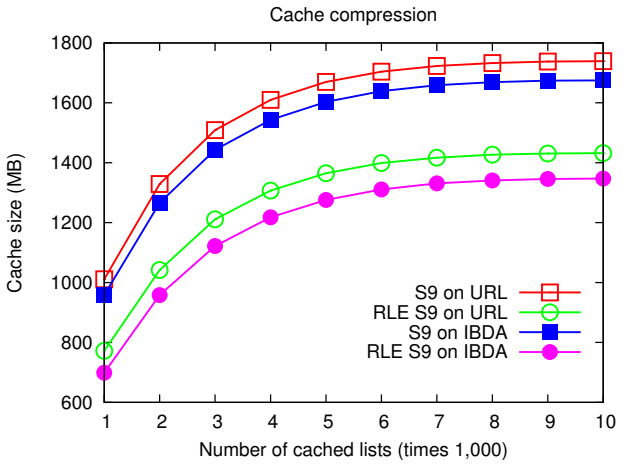
6.5 Inverted List Caching

We study now how the RLE compression methods and the IBDA docID assignment behave when compressing static inverted-list caches [17]. We show that the RLE compression schemes on docIDs assigned with IBDA are more suitable for compressing a small set of inverted lists, rather than using the classic compression schemes on a global docID assignment method, such as URL sorting.

In our experiments, we use as a building block the TREC GOV2 inverted index with docIDs assigned by URL sorting. We use the TREC 2006 query log to compute the frequency of the query terms. We then sort the inverted lists according to this frequency (from most frequent to least frequent). Next, we consider the top-Q lists in this sorting, for $Q = 1$ thousand, 2 thousands, ..., 10 thousands. For each such Q, we apply IBDA on the resulting set of lists, using the order we already mentioned.

Figure 1 shows the space usage as a function of the number of lists considered in each case. We also show a table with the cache hit ratio achieved for every Q. We include just S9 compression, since similar results are obtained for the remaining methods (yet, S9 yields the smallest space usage among all other schemes). As it can be seen, RLE S9 compression yields a considerable reduction of space usage, either on URL and IBDA sorting. For instance, we can conclude that 10 thousand lists compressed with RLE S9 on IBDA sorting use about the same amount of space than 2 thousand lists compressed with S9 on URL. In other words, within the same space used by S9 on URL to achieve a 24% hit ratio, RLE S9 on IBDA is able to achieve a 51% hit ratio.

An important feature that must be noted is that the reduction of space usage degrades as we compress more lists. In our experiments, for 1,000 lists and using RLE S9 on IBDA the reduction is of about 31%. For 10,000 lists, on the other hand, the reduction is of about 23% (recall that for the whole index we obtain a reduction of about 10%, recall Tables 1 and 3).



Lists:	1	2	3	4	5	6	7	8	9	10
% Hit ratio:	13	24	31	37	40	42	44	46	49	51

Figure 1: (Above) Cache space usage as a function of the number of inverted lists compressed, using variants of S9. (Below) Cache hit ratio achieved for the different number of lists.

7. CONCLUSIONS AND FUTURE WORK

We have shown that using our docID assignment scheme IBDA (for focusing the generation of run on a given set of inverted list) and then run-length encoding these lists yields better general performance. This reinforces and improves the results obtained in previous related work [30]. We obtain an improvement of about 10% in space usage compared with the (already very efficient) results from [30]. If we compare our decompression speed with that of [30], we are up to 1.22 times faster if the runs must be explicitly decompressed and up to 4.58 times faster if implicit decompression of runs is allowed. DAAT query processing can be also improved using our approaches: AND queries can be improved by up to 12%, WAND queries by up to 23% and full OR queries by up to 86%. Finally, we have shown that our approaches are useful for the efficient caching of inverted lists, achieving a reduction in space usage of up to 31%.

As future work, we plan to test our approaches to improve the performance of the Block-Max WAND algorithm of [22]. It would be also interesting to use the approach of [25] along with our approaches. Also, it would be interesting to adapt our schemes to graph compression [7]. That is, adapting our IBDA scheme to generate runs in the adjacency lists to then run-length encode these lists.

8. ACKNOWLEDGEMENTS

The authors want to thank the thorough comments from the anonymous referees, which definitely helped to improve the quality of this paper.

9. REFERENCES

[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[2] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Eng.*, 18(6):857–861, 2006.

[3] D. Arroyuelo, S. González, M. Marin, M. Oyarzún, and T. Suel. To index or not to index: time-space trade-offs in search engines with positional ranking functions. In *Proc. of 35th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 255–264. ACM, 2012.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval - the Concepts and Technology Behind Search, Second Edition*. Pearson Education Ltd., Harlow, England, 2011.

[5] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. of 27th European Conference on IR Research (ECIR)*, LNCS 3408, pages 375–387. Springer, 2005.

[6] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of 2002 Data Compression Conference (DCC)*, pages 342–351. IEEE Computer Society, 2002.

[7] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proc. of 13th Int. Conference on World Wide Web (WWW)*, pages 595–602. ACM, 2004.

[8] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 56(18):3825–3833, 2012.

[9] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of 2003 ACM CIKM Int. Conf. on Information and Knowledge Management*, pages 426–434. ACM, 2003.

[10] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[11] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. of 2nd Int. Conf. on Web Search and Web Data Mining (WSDM)*, page 1. ACM, 2009.

[12] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. of 19th Int. Conference on World Wide Web (WWW)*, pages 311–320. ACM, 2010.

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

[14] S. Golomb. Run-length encoding. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[15] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *Proc. of 30th Int. Conf. on Very Large Data Bases (VLDB)*, pages 13–23. Morgan Kaufmann, 2004.

[16] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[17] H. T. Lam, R. Perego, N. T. M. Quan, and F. Silvestri. Entry pairing in inverted file. In *Proc. of 10th Int. Conf. on Web Information Systems Engineering (WISE)*, LNCS 5802, pages 511–522. Springer, 2009.

[18] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[19] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.

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

[21] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1):117–131, 2003.

[22] D. Shuai and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 993–1002. ACM, 2011.

[23] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of 29th European Conference on IR Research (ECIR)*, LNCS 4425, pages 101–112. Springer, 2007.

[24] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. of 27th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 305–312. ACM, 2004.

[25] F. Silvestri and R. Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proc. of 19th ACM Conf. on Information and Knowledge Management (CIKM)*, pages 1219–1228, 2010.

[26] N. Tonellotto, C. Macdonald, and I. Ounis. Effect of different docid orderings on dynamic pruning retrieval strategies. In *Proc. of 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 1179–1180. ACM, 2011.

[27] A. Turpin, Y. Tsegay, D. Hawking, and H. Williams. Fast generation of result snippets in web search. In *Proc. of 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 127–134, 2007.

[28] H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[29] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition*. Morgan Kaufmann, 1999.

[30] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of 18th Int. Conference on World Wide Web (WWW)*, pages 401–410. ACM, 2009.

[31] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surveys*, 38(2), 2006.

[32] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of 22nd Int. Conf. on Data Engineering (ICDE)*, page 59. IEEE Computer Society, 2006.