Hybrid Compression of Inverted Lists for Reordered Document Collections☆

Diego Arroyuelo[a,1,*], Mauricio Oyarzún[b], Senén González[c], Victor Sepulveda[d]

[a]*Dept. of Informatics, Universidad Técnica Federico Santa María, Chile*
[b]*Universidad Arturo Prat – Iquique*
[c]*Software Competence Center Hagenberg GmbH, Austria.*
[d]*Dept. of Computer Science, University of Chile*

Abstract

Text search engines are a fundamental tool nowadays. Their efficiency relies on a popular and simple data structure: *inverted indexes.* They store an *inverted list* per term of the vocabulary. The inverted list of a given term stores, among other things, the document identifiers (docIDs) of the documents that contain the term. Currently, inverted indexes can be stored efficiently using integer compression schemes. Previous research also studied how an optimized document ordering can be used to assign docIDs to the document database. This yields important improvements in index compression and query processing time. In this paper we show that using a hybrid compression approach on the inverted lists is more effective in this scenario, with two main contributions:

- First, we introduce a document reordering approach that aims at generating runs of consecutive docIDs in a properly-selected subset of inverted lists of the index.
- Second, we introduce hybrid compression approaches that combine gap and run-length encodings within inverted lists, in order to take advantage not only from small gaps, but also from long runs of consecutive docIDs generated by our document reordering approach.

Our experimental results indicate a reduction of about 10% – 30% in the space usage of the whole index (just regarding docIDs), compared with the most efficient state-of-the-art results. Also, decompression speed is up to 1.22 times faster if the runs of consecutive docIDs must be explicitly decompressed, and up to 4.58 times faster if implicit decompression of these runs is allowed (e.g., representing the runs as intervals in the output). Finally, we also improve the 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%), outperforming the best existing approaches.

Keywords: Index compression for information retrieval, reordered document collections.

1. Introduction

Inverted indexes are the *de facto* data structure to support the high-efficiency requirements of a text search engine [39, 5, 12, 24, 43]. This includes, for instance, providing fast response to thousands of queries per second, while using as less server memory as possible, among others [15]. 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.

☆A preliminary version of this paper appeared in Proc. of the 36th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR'13).

*Corresponding author. Address: Av. Vicuña Mackenna 3939, Santiago, Chile. Phone: +56 2 2303 7215.

Email addresses: darroyue@inf.utfsm.cl (Diego Arroyuelo), moyarzunsil@unap.cl (Mauricio Oyarzún), sgonzale@dcc.uchile.cl (Senén González), vsepulve@dcc.uchile.cl (Victor Sepulveda)

[1]Funded in part by FONDECYT Grant 11121556, Chile.

1.1. *Inverted Index Compression*

Inverted lists tend to be big, occupying important amounts of memory space. Also, transferring them from secondary storage can take considerable time, degrading query processing time. Hence, lists are 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 [12, see Table 6.9–page 213]. To answer a query, the involved lists (or parts of them) must be decompressed. Therefore, fast decompression is a key issue to support quick answers.

Inverted index compression has been studied in depth in the literature [39, 24, 5, 12]. Usually, docIDs, frequencies, and positions are stored separately, using different inverted-list layers. Thus, each layer can be compressed independently, using the best compression scheme for each case. Even though it is important to compress each of these components, this paper is exclusively devoted to compress docIDs. Frequencies and term positions can be usually compressed using similar techniques. According to [41, 4], the space used by 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 [4]. Besides the space savings, in this paper we are also interested in reducing query processing time.

The most used approach to compress the docIDs of an inverted list I_{w_i} is gap encoding: we first sort the lists by increasing docID, and then represent the list using the differences between consecutive docIDs (minus 1, for technical reasons that will be made clear through the paper). We call DGap these differences. Gap encoding usually generates a distribution with small numbers, in particular for long lists. As we shall see through this paper, many of these DGaps are actually 0s, which correspond to terms that appear in documents whose docIDs are consecutive (recall that we subtract 1 to the difference). We call *runs* the list regions containing consecutive docIDs.

To support efficient searches, compressed inverted lists are logically divided into blocks of, say, 128 DGaps. This allows us skipping blocks at search time, decompressing only the blocks that are relevant for a query. Among the existing compression schemes for inverted lists, we have classical encodings like Elias δ and γ [17] and Golomb/Rice [18], as well as the more recent ones VByte [38], Simple 9 [2], and PForDelta [44] encodings. All these methods benefit from sequences of small integers.

1.2. *Document Reordering*

The assignment of docIDs to a given document database is not trivial, being an important problem in information retrieval [6, 12], and databases [20, 23]. This task—usually known as *document reordering*—sorts the documents in $\mathcal{D}$, to then assign the docIDs following this order. For instance, sorting the documents according to their URLs is a simple and effective method [32]. These are called *ordered document collections* and will be the focus of this paper. Document reordering is not always feasible, as discussed in [41]. However, there are still 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. Although the original problem can be formulated as an instance of the TSP problem (an hence, it is NP-Complete) [30], there are several heuristic approaches that achieve a better compression performance [7, 8, 30, 32, 33], and in general a much better inverted index performance [41, 35]. Improvements of up to 50% both in space usage and *Document at a Time* (DAAT from now on) query processing have been reported [41, 35, 26]. This problem has been also studied for databases and bitmap indexes [20, 23], and for the particular case of e-Commerce [28], among others, with similar conclusions.

A remarkable feature of ordered document collections is that the number of consecutive docIDs within inverted lists is increased, which explains the improvement in space usage. For instance, after reordering the well-known TREC GOV2 document collection, about 60% of the postings correspond to consecutive docIDs, whereas a random ordering yields just 11% of consecutive docIDs.

But, more importantly, these consecutive docIDs actually form long *runs* in the inverted lists. These runs of consecutive docIDs are runs of 0s when gap encoded. Rather than regarding these 0s separately in the encoding, *we propose to deal with the runs as a whole.* Having runs of equal symbols in a sequence allows us to use, for instance, run-length encoding [18]: we simply encode a run writing its length. For inverted lists, however, using run-length encoding would not be as effective as for other applications, as there are only runs of 0s: other values rarely form long runs. To tackle this issue, we introduce compression approaches that mix gap encoding along with run-length encoding. The idea is that run-length encoding is used for the segments where runs occur, leaving gap encoding for the remaining portions of the lists.

1.3. Previous Work for Compressing Small DGaps/Runs

The following are the main ways to take advantage of small DGaps and runs in inverted lists, namely:

1. Compression approaches that have been particularly tuned to compress small DGaps, like the ones by Yan et al. [41]. In particular, their OptPFD approach is able to encode long runs of consecutive docIDs using 1 bit per docID.
2. Compression approaches like *Interpolative Encoding* [25], which is suitable to compress lists of consecutive values. Unlike the previous approach, Interpolative encoding is able to represent long runs of consecutive docIDs implicitly. Unfortunately, the decoding process in this case is slow, hence the query processing performance one can obtain by using Interpolative encoding is not competitive [12, 41].
3. Compression approaches like *Interval Encoding* [9], which is able to compress the runs in the 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 this approach is effective for web graph compression, it has two potential drawbacks for information retrieval. First, the separate storage of the docIDs (i.e., intervals and residuals) is not adequate to support efficient DAAT query processing. Second, after removing the intervals from the list, bigger DGaps are generated, which is similar to the effect seen in [22]: unfortunately, this worsens the compression ratio achieved.
4. Compression approaches like the one proposed in Anh and Moffat [3]. They suggest that their method can compress runs of consecutive docIDs in inverted lists. Claude et al. [14] applied this idea to compress inverted indexes for versioned document databases.
5. Compression approaches like the one by Ottaviano and Venturini [26]. They modify the original Elias-Fano encoding, in order to take advantage of the small DGaps generated by consecutive docIDs. However, although being competitive at query time, their approach is most optimized for space usage.
6. Compression approaches like *VSEncoding* [34], which computes the optimal block partitioning to improve both compression ratio and decompression speed. We hope that this approach can be used along with the techniques proposed in this paper to implicitly compress runs of consecutive docIDs.

1.4. Our Contribution

We study how the compression of inverted indexes and DAAT query processing are affected by the runs of consecutive docIDs that are generated in ordered document collections. We show that the hybrid gap + run-length encoding of inverted lists offers competitive space/time trade-offs for inverted indexes of ordered document collections. In particular, we propose to use run-length encoding for the list regions containing consecutive docIDs, and gap encoding for the remaining list regions. To obtain a successful approach, we carry out the following:

- Propose a document reordering method that aims at generating not only small DGaps in the inverted lists, but also runs of consecutive docIDs. This is relevant for the success of the compression approaches that will be mentioned in the next item. Moreover, our reordering approach allows one to choose a particular subset of inverted lists, corresponding to certain terms of interest. Then, the reordering process will focus on the documents that contain these terms. Typically, these subsets corresponds to big lists that need to be transferred or, alternatively, the most frequently accessed lists (a better compression could allow one to keep these lists cached in main memory), just to name a few. This allows one to choose the inverted lists on which to improve the performance (i.e., compression and query processing time), which adds value to our approach.

- Propose hybrid compression approaches that are able to run-length encode these runs, as well as using gap encoding for the remaining parts of the list.

Overall, our compression approaches support efficient DAAT query processing. We aim at the efficient space usage of approaches (2) and (3) above, but this time supporting efficient query processing time. We also show that our approaches outperform approach (1) in practice, which suggests that a hybrid compression approach could be more efficient in these cases. When compared to approach (5) above, we show that our results offer competitive space/time trade-offs.

2. Previous Concepts

2.1. Inverted List Compression

2.1.1. Gap Encoding

Gap encoding is one of the most-used approaches to compress inverted lists. Let $d_i[1..n_i]$ be the document identifiers corresponding to inverted list I_{w_i}, stored in increasing way. Hence, we replace $d_i[1]$ with $d_i[1] - 1$, and $d_i[j]$ with $d_i[j] - d_i[j-1] - 1$, for $j = 2, \ldots, n_i$. These gaps (called DGaps in this paper) are then compressed using any compression approach that is able to encode small integer values using less space. Some typical compression schemes are Elias δ and γ encodings [17], and Golomb/Rice encoding [18]. Even though these usually achieve a good compression ratio, their slow decompression speed makes them less practical at query time [12]. Practitioners usually prefer compression schemes that are faster to decompress, a key feature to support fast queries, at the price of using slightly more space. We review some of these approaches next.

2.1.2. Compression Schemes

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

Variable Byte. Variable-byte (VByte for short) encodes an integer using a variable number of bytes. 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 is called continuator. If the continuator is "1" it means that the current byte is not enough to encode the current integer and hence the encoding must use the next byte. Otherwise, if the continuator is "0" then the current byte is the last used by the encoding. In other words, to encode a number in VByte, we try to accommodate its binary encoding into 7 bits. If so, we store the encoding in 7 bits and mark the continuator with a "0". Otherwise the least-significant 7 bits of the integer are stored in a byte with continuator "1" and proceed with the remaining bits, until no extra bytes are needed to store the encoding. As the encoding is byte-aligned, it can be decoded faster.

Even though VByte uses the exact number of bytes to accommodate the binary encoding of the number, it is important to note that 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).

Finally, the decoding process of VByte can be carried out efficiently using bit-wise and arithmetic operations.

Simple 9. The VByte scheme encodes each integer individually. In the best scenario, VByte is able to encode 4 integers within a 32-bit machine word. However, if the integers to encode are small (e.g., able to be encoded with 1 bit) we waste space. The basic idea of the Simple 9 approach [2] (S9, for short) is to take several consecutive integers in the sequence and try to fit them within a 32-bit machine word, encoding each integer within 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 a so-called S9 *word* of 32-bit. In each S9 *word*, 4 bits are used as a header to define the S9 *case*, which indicates how many bits will be used to encode the integers, and also how many integers will be stored in the S9 *word*. The remaining 28 bits store the integers themselves, encoded in binary.

There are 9 possible ways of dividing 28 bits into chunks of equal size. The meaning of the cases is as follows:

(C1) : 1 chunk of 28 bits, indicated by the 4-bit header 0000.
(C2) : 2 chunks of 14 bits, indicated by the 4-bit header 0001.
(C3) : 3 chunks of 9 bits, plus 1 unused bit, indicated by the 4-bit header 0010.
(C4) : 4 chunks of 7 bits, indicated by the 4-bit header 0011.
(C5) : 5 chunks of 5 bits, plus 3 unused bits, indicated by the 4-bit header 0100.
(C6) : 7 chunks of 4 bits, indicated by the 4-bit header 0101.
(C7) : 9 chunks of 3 bits, plus 1 unused bit, indicated by the 4-bit header 0110.
(C8) : 14 chunks of 2 bits, indicated by the 4-bit header 0111.
(C9) : 28 chunks of 1 bit, indicated by the 4-bit header 1000.

During the encoding process, we try to fit as much integers as possible within the 28 available bits of an S9 *word*. 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 where only one 28-bit integer can be stored. In the header of the S9 *word* we store the particular case used for it.

Example 2.1. To encode the integers 98, 112, 117 and 121, notice that each of them can be encoded in binary using 7 bits. Hence, we can use C4. The resulting S9 *word* is shown in Figure 1.

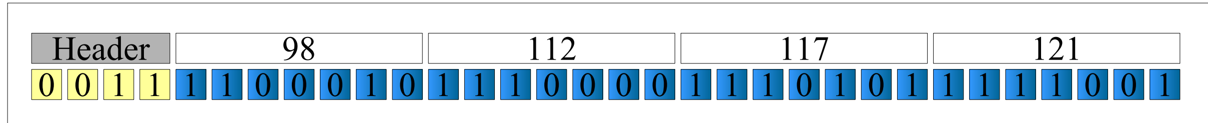

Figure 1: Schematic representation of an S9 *word* encoding the group of integers 98, 112, 117 and 121 (above), and the corresponding binary encoding (below).

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. The main feature of this method that makes it highly efficient at decoding time is that all the integers in an S9 *word* are encoded 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 approach is S16 [42], where all the 16 possible cases that we can have with headers of 4 bits are defined, in order to be able to catch the most common cases that arise in inverted lists.

PForDelta. The PForDelta encoding [44] 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 stores information about the compression used in the block, e.g., the value of b. 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 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 2.2. A PForDelta block that encodes 128 integers is shown in Figure 2. The exceptions are 78, 110, 160 and 91.

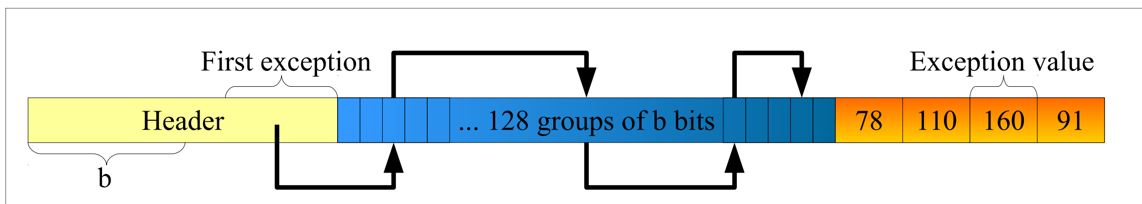

Figure 2: PForDelta block, encoding a group of integers, with exceptions $78, 110, 160$ and 91.

To decompress a block, we first take b from the header, and invoke a specialized function that obtains the b-bit 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). The remaining bits in the header can be used to save information that is particular to each implementation. PForDelta has shown to be among the most efficient compression schemes in practice [41], achieving a high decompression speed.

2.1.3. Inverted List Layout for Document-at-a-Time Query Processing

Given a query consisting of q terms $w_{i_1}, \ldots, w_{i_q}$, corresponding to the inverted lists $I_{w_{i_1}}, \ldots, I_{w_{i_q}}$, Document-at-a-Time (DAAT) processes simultaneously—and from left to right—each of these inverted lists. This process is usually carried out using the function `nextGEQ`(I_{w_j}, d), which yields the smallest docID d' in inverted list I_{w_j} such that $d \leq d'$. Through the query process, `nextGEQ` is invoked with increasing values of d each time, hence 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 efficient DAAT, inverted lists are typically stored using a block layout to support efficient access: 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 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: these are not decompressed, saving time. Once we arrive at the block that potentially stores d, we fully decompress the block 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.

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 [7, 8, 30, 32, 33]. This generates smaller DGaps in the inverted lists, and hence better compression. It is important to note that document reordering is not always feasible [41]. In particular, when global measures of page ranking (e.g., PageRank [10] and Hits [21]) are used for early termination. However, some applications benefit from document reordering.

Previous work [41] studies the performance of different compression schemes for ordered document collections. One of their main conclusions is that some compression schemes are not suitable for these cases, e.g., PForDelta, since most DGaps are small numbers. 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 when b is too small.

The following compression alternatives were introduced to amend this problem [41].

New PForDelta. This approach (NewPFD from now on), supports any value $b \geq 1$ without adding extra exceptions. The trick is to use the b-bit slots to store the lower b bits of the exceptions, while the remaining bits are stored in a separate space of memory (and compressed using S16 [41]). The offsets (to the next exception) are also stored in a separate array in compressed form. At decompression time, the original offsets are obtained from their compressed form, and are used afterwards to reconstruct the exceptions.

Optimized PForDelta. This alternative (OptPFD from now on), allows one 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 [41] is that, given a target decompression speed, NewPFD and OptPFD have a better space usage than PForDelta.

Besides improving the space usage of inverted lists, existing document reordering methods, in general, reduce the query processing time [41, 35]. The reason is that, after reordering, documents that are relevant to a query tend to group into a few inverted-list blocks, hence less blocks need to be decompressed at query time. In other words, the locality of access to inverted list blocks is improved, reducing the number of blocks that need to be decompressed. Improvements of up to 50% in query processing time have been reported [41].

In this paper we propose a different approach: rather than making the gap-encoding compressors more efficient to handle small integers [41], we consider a hybrid encoding of the inverted lists, such that long runs of consecutive docIDs can be handled efficiently. We will show that this allows us to achieve a better performance in many cases.

3. Document Reorder and the Generation of Runs

Inverted list compression based on DGap encoding benefits from distributions with small DGaps. Document reordering (as seen in Section 2.2) yields smaller DGaps. For instance, in the inverted index of the TREC GOV2 collection and the original document order (we will call "non-ordered collection" to this order), just 10.75% of DGaps have value 0 (i.e., correspond to consecutive docIDs). If, on the other hand, we sort the documents by URLs [32], we obtain 60.30% of DGaps with value 0. This can be seen in Figure 3, which shows the distribution for the DGap values for different document orders. Thus, the ordered case generates many 0 DGaps [16].

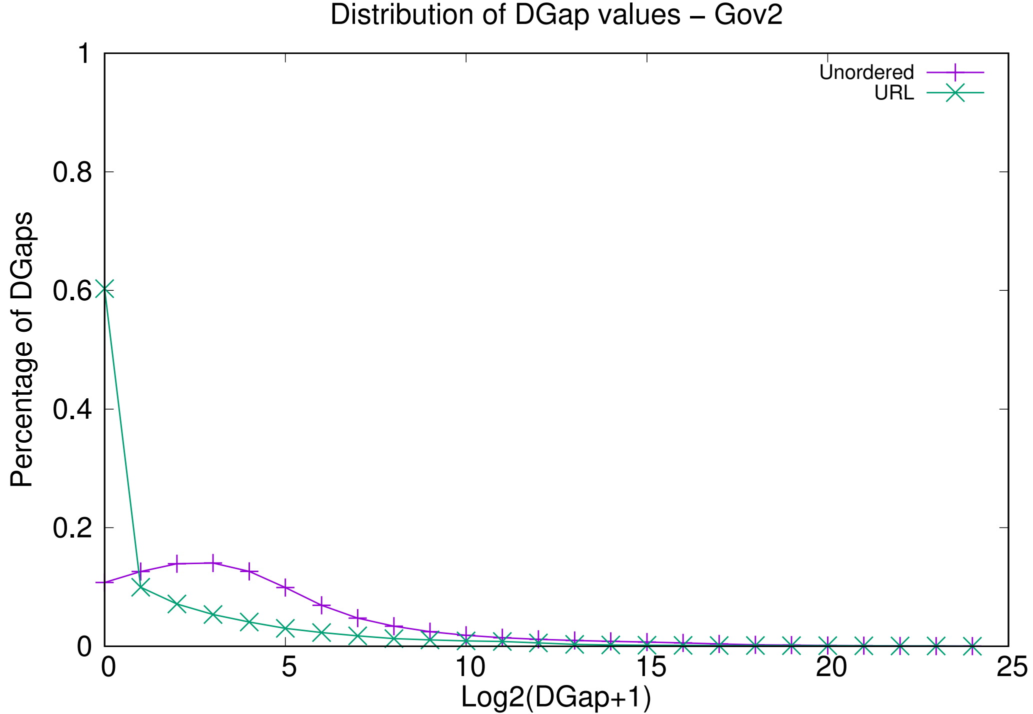

Figure 3: Distribution of gap values for the GOV2 collection.

Actually, we note that many of these 0s are grouped into long runs. This can be seen in Figure 4, which shows the distribution of run lengths, comparing the non-ordered collection with `URL` order. This fact can be used to improve compression: rather than regarding these DGaps separately [41], we propose to deal with the runs as a whole. Runs can be encoded more efficiently, e.g., using *run-length compression* [18, 29].

Next, we develop document reordering methods that aim at generating not only small DGaps, but also long runs in the inverted lists.

3.1. Document Reorder Based on Intersections

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 renumerated 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 $d_1 \to 1$, $d_2 \to 2$, $\ldots$, $d_l \to l$. For each such d_i, we set $F = \{(d_1, 1), (d_2, 2), \ldots, (d_l, l)\}$. Notice how this transforms I_{i_1} into a single run when representing the list as DGaps.

Next, we go on to renumerate the second inverted list I_{i_2}, with the restriction that every docID i such that $(i, j) \in F$ (for a given j) cannot be further renumerated during the process. These are called *fixed* docIDs, and the rationale is to avoid breaking the runs that we have already created. 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.

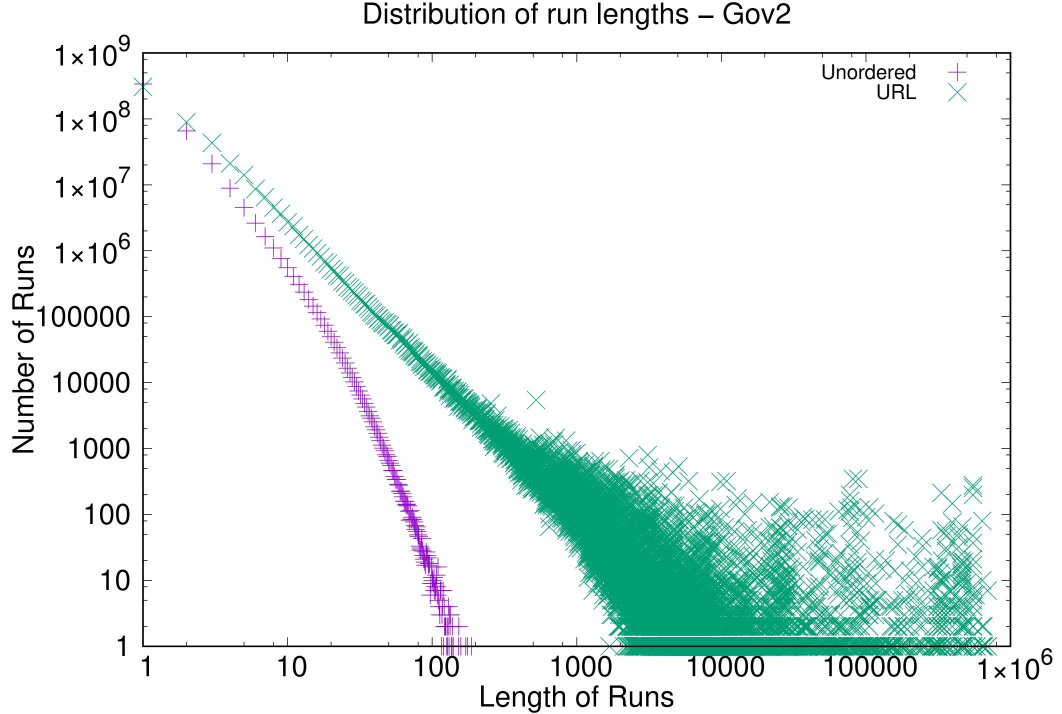

Figure 4: Distribution of run lengths for the GOV2 collection.

Example 3.1. For instance, let us consider $I_{i_1} = \langle 10, 30, 65, 66, 67, 70, 98 \rangle$ and $I_{i_2} = \langle 20, 30, 66, 70, 99, 101 \rangle$. The first step re enumerates I_{i_1} assigning $10 \to 1$, $30 \to 2$, $65 \to 3$, $66 \to 4$, $67 \to 5$, $70 \to 6$, $98 \to 7$. Now we renumerate I_{i_2}, having into account that docIDs 30, 66 and 70 are now fixed. Hence, we assign $20 \to 8$, $99 \to 9$, $101 \to 10$. 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 renumerated in any order when processing I_{i_1}.

We can fix this problem as follows. We instead start by re enumerating 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 renumerate 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.

Example 3.2. For the previous example, we start re enumerating the intersection $I_{i_1} \cap I_{i_2}$ as $30 \to 1$, $66 \to 2$, $70 \to 3$. Then, we renumerate the remaining elements in I_{i_1} as $10 \to 4$, $65 \to 5$, $67 \to 6$, $98 \to 7$. Finally, we renumerate the remaining elements in I_{i_2} as $20 \to 8$, $99 \to 9$, $101 \to 10$. 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}. Notice that since these d documents are stored in every list $I_{i_1}, \ldots, I_{i_j}$, we are actually creating a run in all of them.

Next, 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 I_{i_1} and I_{i_2};
- $I_{i_1} \cap I_{i_2} \cap I_{i_3}$ in I_{i_3};
- $\ldots$;
- $I_{i_1} \cap \cdots \cap I_{i_j}$ in I_{i_j}

are transformed into respective runs in each of these lists after the first step of IBDA. This process not only creates runs in the inverted lists, but also gives us a way to precompute intersections. In order to take full advantage of this, we must set a suitable initial order for the lists in $\mathcal{L}$: if we want to transform the intersection between relevant inverted lists into a run, these lists must be consecutive in $\mathcal{L}$. This can yield to query-time improvements (see Section 5).

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.

3.2. Experimental Evaluation

In our experiments, we use the following document orders to assign the docIDs to the TREC GOV2 collection (of 25.2 million documents, and a vocabulary of 32.8 million terms), and the Wikipedia (dump of August 2016, with 8.6 million documents, and a vocabulary of 6.5 million terms): (1) the original order given to the documents in the collection ("Unordered" from now on); (2) the URL sorting ("URL" from now on); (3) the IBDA method introduced in Section 3.1.

3.2.1. A Practical Implementation of IBDA

We found out that IBDA performs better on inverted indexes of already-ordered document collections. Thus, we apply IBDA to the inverted index constructed for URL. In our implementation of IBDA, we sort the inverted lists according to the following procedure (recall from Section 3.1 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 w_i and w_j are the most-frequently-queried pair of terms, 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.

3.2.2. Analysis of Results

Figure 5 shows the distribution of DGap values for the three document orders we consider, for GOV2 (left) and the Wikipedia (right). The result is that for GOV2, IBDA slightly increases the amount of consecutive docIDs in the lists. For Wikipedia, on the other hand, the amount of consecutive docIDs is roughly duplicated. It is already known that it is hard to generate consecutive docIDs in this case [16].

Finally, URL +IBDA yields a distribution of runs that is similar to URL, except that the former generates some (few) very long runs. Although the percentage of 0 DGaps that is introduced by applying IBDA on URL is marginal (about 2%, as seen in Figure 5), the better run distribution yielded by IBDA will be key for reducing space usage and query processing time (as we shall see later).

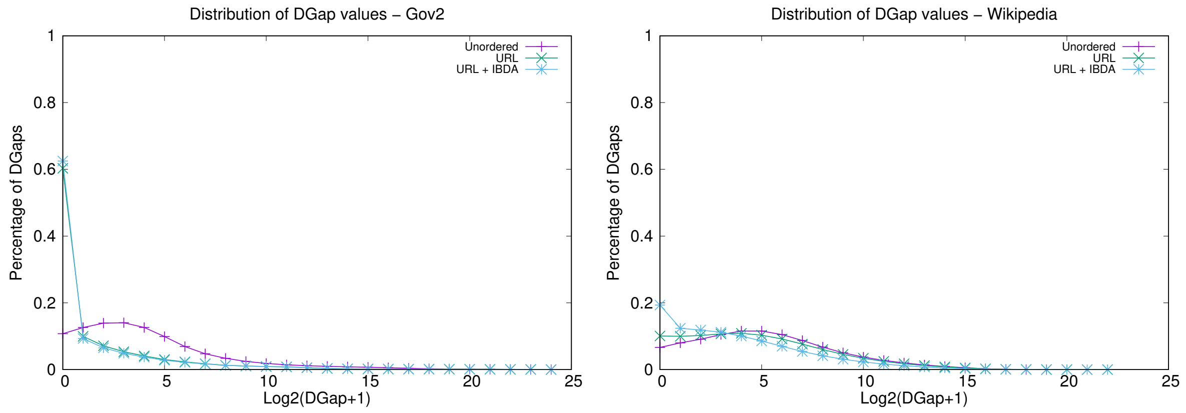

Figure 5: Distribution of DGap values for the different document reordering methods, for GOV2 and Wikipedia inverted indexes.

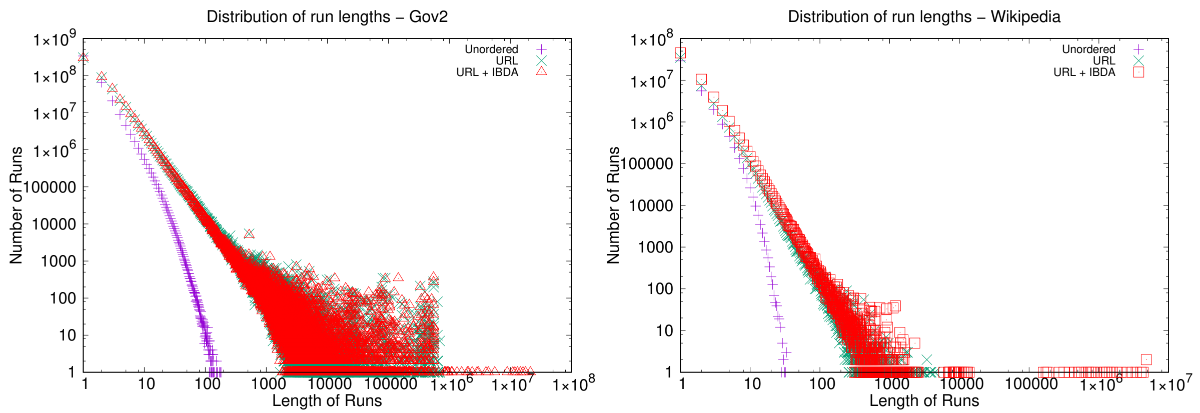

Figure 6: Distribution of run lengths for `Unordered`, `URL`, and `URL` +`IBDA` document reorderings, for GOV2 and Wikipedia inverted indexes.

4. Hybrid Compression of Inverted Lists

Given inverted lists corresponding to an ordered document collection, we propose to deal with the runs of consecutive docIDs, rather than dealing with the small DGaps separately. In some applications (Such as B&W image compression), run-length encoding is an effective way to deal with runs of repeated values: we replace a run of a certain value by storing only its length and the value that forms the run.

However, using run-length compression would not be as effective for encoding inverted lists because, in practice, only 0-DGaps tend to form runs. To tackle this problem, we propose a hybrid approach that mixes run-length encoding (for the list regions storing consecutive docIDs) and gap encoding (for the remaining parts of the lists). We adapt some of the most effective compression schemes in order to achieve this, aiming at efficient query processing time and improved compression ratios, outperforming previous work in many cases [41, 26].

Unlike previous sections, for our hybrid approaches we need to redefine the way in which DGaps are stored. That is, if $d_i[1..n_i]$ are the document identifiers corresponding to an inverted list I_{w_i}, in increasing order, we store $d_i[1]$ as it is, whereas $d_i[j]$ is replaced with $d_i[j] - d_i[j-1]$, for $j = 2, \ldots, n_i$. In other words, we do not subtract 1 to each DGap as in previous sections. As a result, the smallest DGap we can store is 1, and corresponds to two consecutive docIDs. A run of consecutive docIDs becomes, then, a run of 1s when gap encoded. This decision could yield a slight increase in space usage, in particular, when the list has just a few runs of consecutive docIDs. However, it is essential for our approaches.

4.1. A Hybrid Version of VByte

4.1.1. Tuning VByte to Handle Small DGaps

We introduce first an alternative implementation of the original VByte scheme, which will be our building block. The idea is to take advantage, at decompression time, of several consecutive small DGaps (which is rather common for ordered collections). 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 (as we will see in our experiments). This is similar to the VarInt schemes presented by Dean [15]

4.1.2. H-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 $\underline{0}0000000$ as the special mark. The idea is to replace $\ell \geq 3$ consecutive 1s in an inverted list by the $\underline{0}0000000$ mark, followed by the VByte encoding of ℓ. We call H-VByte this scheme, which is able to encode small runs (of length ≥ 3). Indeed, this will be the only method in this paper that besides being able to encode longer runs, will be also able to encode runs of length $3 \leq \ell \leq 27$. We restrict the minimum length to be 3, because this yields the same space as for runs of length 2 (using 2 bytes in both cases), yet the former has better decompression performance.

Example 4.1. Figure 7b shows an example of H-VByte compression of the sequence shown in Figure 7a.

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

4.2. A Hybrid Version of S9

Next, we define S18, a hybrid variant of S9 that combines gap and run-length encodings within a single inverted list. It is important to recall that we store DGaps ≥ 1. Hence, 1 is the only value that can be stored in chunks of 1 bit. This means that the S9 case of 28 chunks of 1 bit actually stores 28 1s. We use this fact in what follows.

S18 will run-length encode consecutive 1 DGaps. A first approach is to compress the sequence of DGaps using the original S9 algorithm. Next, we traverse the S9 words obtained, and replace $\ell > 1$

$$98, 112, 5, 68, \underbrace{1, \ldots, 1}_{\text{Run of 28 1s}}, 13, 1, 9, 1, 4, 1, 8.$$

(a) A sequence of 39 DGaps to be compressed.

$$\underline{0|98} \quad \underline{0|112} \quad \underline{0|5} \quad \underline{0|68} \quad \underbrace{\underline{0|0} \quad \underline{0|28}}_{\text{Run of 28 1s}} \quad \underline{0|13} \quad \underline{0|1} \quad \underline{0|9} \quad \underline{0|1} \quad \underline{0|4} \quad \underline{0|1} \quad \underline{0|8}$$

(b) H-VByte representation of the above sequence. Each of the 13 bytes in the encoding is shown underlined. The VByte continuator bit is shown to the left of "|" in each byte.

$$\underbrace{98, 112, 5, 68}_{\text{C4}}\underbrace{13, 1, 9, 1, 4, 1, 8}_{\text{C12}}.$$

(c) S18 representation of the sequence. Notice that C12 implicitly encodes the run of 28 1s, hence the encoding consists only of 2 words (8 bytes).

Figure 7: Hybrid encoding of a sequence of 39 DGaps.

consecutive S9 words encoding 28 1s, by a single S9 word storing the length ℓ in 28 bits. In other words, we run-length encode the run of consecutive 1s. To do this, we need to add an extra case to S9 in order to properly decode it. This simple idea has been already suggested in a previous work [3], as well as used to compress inverted indexes of versioned document databases [14].

However, short runs cannot be compressed in this way. For instance, single S9 words that encode 28 1s still need to be explicitly encoded. We define next an approach that, besides run-length encoding the runs on consecutive 1s, avoids storing single S9 words that encode 28 1s, saving space (as short runs are in many cases more common than longer ones).

We introduce extra cases to define S18. First, the original S9 cases are included. Second, we add a case to encode consecutive S9 words that encode 28 1s, as explained before. Finally, we add 8 extra cases to encode words of 28 1s that are followed by a different case. The total is 18 cases. We still have a 4-bit header, hence only at most 16 cases can be encoded. However, we arrange the cases in such a way that allows us to have 18 cases, being able to properly decode them.

We start by defining the following S18 cases:

(C1) 1 chunk of 28 bits, indicated by the 4-bit header `0000`;
(C2) 2 chunks of 14 bits each, indicated by the 4-bit header `0001`;
(C3) 3 chunks of 9 bits each, with 1 unused bit, indicated by the 4-bit header `0010`;
(C4) 4 chunks of 7 bits each, indicated by the 4-bit header `0011`;
(C5) 7 chunks of 4 bits each, indicated by the 4-bit header `0100`;
(C6) 9 chunks of 3 bits each, with 1 unused bit, indicated by the 4-bit header `0101`;
(C7) 14 chunks of 2 bits each, indicated by the 4-bit header `0110`.

We add now the cases where a 28 1-bit word precedes another (different) case. In other words, we encode two consecutive S9 words into just one S18 word. This is the only way in which these moderately-short runs can be compressed by S18. Indeed, these kinds of runs are actually the most frequent ones in our experimental results. The key here is that a 28 1-bit word only can store 28 1s. The cases are defined as follows:

(C8) 28 chunks of 1 bit each, followed by C1, indicated by the 4-bit header `0111`;
(C9) 28 chunks of 1 bit each, followed by C2, indicated by the 4-bit header `1000`;
(C10) 28 chunks of 1 bit each, followed by C3, indicated by the 4-bit header `1001`;
(C11) 28 chunks of 1 bit each, followed by C4, indicated by the 4-bit header `1010`;
(C12) 28 chunks of 1 bit each, followed by C5, indicated by the 4-bit header `1011`;
(C13) 28 chunks of 1 bit each, followed by C6, indicated by the 4-bit header `1100`;
(C14) 28 chunks of 1 bit each, followed by C7, indicated by the 4-bit header `1101`;
(C15) 28 chunks of 1 bit each, followed by 5 chunks of 5 bits each, indicated by the 4-bit header `1110`;

We finalize the definition of S18 by adding the following 3 cases. The case indicators for them will share the 4-bit prefix `1111`. We need to add extra bits to differentiate them, as follows:

- (C16) 28 chunks of 1 bit each. This case is used when an inverted list ends with a word storing 28 1s, and it is indicated with the 5-bit header `11111`. Since we are encoding 1s in this case, there is no need to explicitly store them. Hence, it is safe to use 1 bit from the data part as the 5th bit of the header.
- (C17) 5 chunks of 5 bits each (with 3 unused bit), indicated by the 6-bit header `111100`. This is an original S9 case that was postponed because of the 3 unused bits. We use 2 of them to encode the header, and leave 1 unused bit.
- (C18) The last case is for $2 \leq \ell \leq 2^{26}$ consecutive words encoding 28 1s each. This case is indicated by the 6-bit header `111101`, leaving 26 bits to encode ℓ in binary.

These variable-length headers can be uniquely decoded efficiently using a `switch` statement in C/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. If it is `0`, we need to check an extra bit to determine if it corresponds to cases C17 or C18. Otherwise it corresponds to C16.

Example 4.2. Figure 7c shows an example of S18 compression of the sequence shown in Figure 7a.

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 C18; if instead we have one such word followed by a different case, we replace it by the corresponding case 8 to 15.

4.3. A Hybrid Version of PForDelta

We define now H-PFD, which adds run-length-encoding capabilities to PForDelta, with minor changes to the original scheme. We define two kinds 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 an unused bit in the header. At decompression time, the flag is checked to see whether one must decompress a normal block or a run.

Similarly to S18, this scheme is unable to encode relatively short runs. We use 32 bits (the header) to represent the run length. Hence, run blocks encode runs of length ≥ 32, thus using at most one bit to encode each 1 in the run.

4.4. 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 hybrid compression schemes using C++. We compile our codes with `g++`, with optimization flag `-O5`. We compare our results with the most efficient and practical compression schemes, namely PForDelta, VByte, S9, and their variants. It is important to recall that these approaches are provided by search tools such as Lucene [2] and Terrier [3] [27]. For PForDelta, NewPFD, and OptPFD we use the highly-efficient implementations from [41]. We base our implementation of H-PFD on OptPFD. VByte and S9 implementations are from us. For VByte, S9, PForDelta, NewPFD and OptPFD, we subtract 1 to each DGap in the index, so they start from zero. We consider inverted lists of length ≥ 128 in our experiments. The total number of docIDs stored in the inverted index is 6,130,535,429.

4.4.1. 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:

[2] `https://lucene.apache.org`
[3] `http://www.terrier.org`

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. The total number of docIDs is 6,130,535,429. The space in bits per docID is shown in brackets.

Compression Scheme	Reorder Method		
	`Unordered`	`URL`	`IBDA`
Interpolative	5,507 [**7.19**]	2,712 [**3.54**]	2,641 [**3.45**]
VByte	7,526 [**9.82**]	6,726 [**8.78**]	6,754 [**8.81**]
S9	6,687 [**8.73**]	3,777 [**4.93**]	3,735 [**4.87**]
OptPFD	6,348 [**8.28**]	4,600 [**6.01**]	4,504 [**5.88**]
H-VByte	7,418 [**9.68**]	3,861 [**5.04**]	3,743 [**4.88**]
S18	6,687 [**8.73**]	3,455 [**4.51**]	3,392 [**4.42**]
H-PFD	6,384 [**8.33**]	4,264 [**5.56**]	4,137 [**5.39**]

- The space usage of interpolative encoding, S9, and OptPFD is slightly improved when using `IBDA` order, rather than just sorting by `URL`.
- The hybrid compression schemes on docIDs assigned by `URL` and `IBDA`, reduce considerably the space usage, compared with the corresponding normal schemes. In particular, it is worth to note that H-VByte has a 44.58% reduction. This is because relatively-short runs are rather frequent in inverted indexes and, unlike the other schemes, H-VByte is able to encode runs of length ≥ 3. Also, unlike the original VByte, H-VByte is able to use less than 8 bits per docID (when there are runs in the sequence).
- Disregarding interpolative encoding, the smallest space usage is achieved by S18 on docIDs assigned by `IBDA`. The improvement over S9 on docIDs assigned by `URL` (the previously-known best performer [41]) is of about 10.19%. If we only consider `URL`, using S18 instead of S9 yields a saving of about 8.52%.
- Finally, S18 uses between 1.25 and 1.28 times the space of interpolative encoding, depending on the docID assignment scheme used.

4.4.2. 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. Our hybrid compression methods *implicitly* decompress the runs. That is, we write in the output the run length, plus a mark indicating that this number must be interpreted as a run length.

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		
	`Unordered`	`URL`	`IBDA`
Interpolative	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
H-VByte	446.58	1,313.03	1,988.86
S18	779.52	1,812.22	2,691.09
H-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 interpolative (by 49.67%), S9 (by 21.93%) and VByte (by 10.81%). This is remarkable, and adds independent interest to `IBDA`.

- The decompression performance of our VByte variant (recall Section 4.1) is faster than the original implementation of VByte (which in our tests decompresses 571 million docIDs per second for `Unordered`, 576 million for `URL`, and 594 million 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 [15] 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 [36].
- For `URL` order, S18 is 1.84 times faster than S9, whereas H-PFD is 2.36 times faster than OptPFD. For `IBDA`, on the other hand, S18 is 2.24 times faster (compared to S9), whereas H-PFD is 4.61 times faster than OptPFD. Finally, note that `Unordered`, S18 is slightly faster than S9 (1.04 times), while using the same space (see Table 1).

The results in Table 1 and Table 2 indicate that for `IBDA`, our compression approaches H-VByte, S18, and H-PFD offer attractive space vs. decompression-time trade-offs. We can also conclude that `IBDA` by itself is able to improve decompression speed of the original compression approaches: using S9 on `IBDA` order, decompression speed is improved by about 22% when compared to `URL` order. However, using our hybrid compression methods on `IBDA` order is the key to obtain the remarkable speedups that we mention above.

5. Runs and DAAT Query Processing

Document-at-a-Time (DAAT) and *Term-at-a-Time* (TAAT) are the usual ways to support efficient query processing in inverted indexes [12]. DAAT is generally faster and uses less memory than TAAT. Next we adapt the traditional DAAT query processing to efficiently handle the runs of consecutive docIDs in the lists, which are run-length encoded by the compression approaches proposed in this paper.

5.1. DAAT and Implicit Decompression of Runs

As in Section 4.4.2, we do implicit decompression of runs, this time to support DAAT query processing. That is, rather than explicitly writing in `Buffer` the docIDs in a run, we write the special mark `0` in the corresponding position of `Buffer`, followed by the length of the run. Thus, the time needed to decompress the whole run will be that needed to decompress just its length (i.e., a single integer).

Assume that, at query time, we invoke `nextGEQ`(I_{w_j}, d) and that the block that is candidate to store d has been already decompressed into `Buffer`. Suppose that, looking for d, we reach position j of `Buffer` and `Buffer`$[j] = 0$ holds. That is, we have reached a run of length $\ell =$ `Buffer`$[j+1]$. Let d' be the docID corresponding to position $j-1$ in `Buffer`. Hence, 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 (as it does not contain d) and go on from position $j+2$ in `Buffer`. Otherwise, if $d'+\ell \geq d$ holds, the sought docID d 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 $d' = d$, which will indicate to the subsequent `nextGEQ` invocation that we have processed up to docID d.

We maintain the values of p and d' for the next search in the list. Assume that afterwards 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 from position $j+2$ in `Buffer`. Otherwise, d'' also lies within the run, hence we act as before, updating d' and p.

5.2. Redefining the Block Layout of the Lists

To support DAAT query processing, we need to redefine the block structure of the inverted lists. The objective of dividing an inverted list into blocks of fixed size is that only the relevant blocks would need to be decompressed at query time, which saves time. Therefore, a block cannot store too many DGaps, as this would worsen the decompression performance. Blocks of size 128 seem to be a good compromise in practice. However, in our case the inverted lists can store runs, which do not need to be decompressed explicitly. Actually, the cost of decompressing a run is that of decompressing a single integer. Also, once it has been decompressed, a run can be handled efficiently by the `nextGEQ` function (as we explained before).

We define blocks of fixed size, say 128 integers per block. However, many of these integers represent runs, hence a block can store a variable number of docIDs. This will allow us to reduce the number

of blocks in the lists, except for H-PFD. As a result, we would need to store less block headers, thus reducing the space usage. Also, this will allow us to reduce the number of blocks decompressed at query time, improving the query performance.

5.3. 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.

5.3.1. Space Usage

Table 3 shows the space usage of the TREC GOV2 inverted index. We include the space required by the block headers. In order to compare with previous work, the results from Yan et al. [41] correspond to

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		
	Unordered	URL	IBDA
Interpolative	6,551 [**8.55**]	3,756 [**4.90**]	3,685 [**4.81**]
VByte	8,264 [**10.78**]	7,464 [**9.74**]	7,497 [**9.78**]
S9	7,524 [**9.82**]	4,601 [**6.00**]	4,572 [**5.97**]
OptPFD	6,835 [**8.92**]	5,087 [**6.64**]	5,105 [**6.66**]
H-VByte	8,448 [**11.02**]	4,626 [**6.04**]	4,485 [**5.85**]
S18	7,524 [**9.82**]	4,147 [**5.41**]	4,091 [**5.34**]
H-PFD	7,267 [**9.48**]	5,003 [**6.53**]	4,850 [**6.33**]

URL order, and the compression approaches VByte (using 7,464 MB), S9 (using 4,601 MB), and OptPFD (5,087 MB). By using our compression approaches and the IBDA order, we are able to reduce the space usage by 39.91% with H-VByte (4,485 MB), 11.08% with S18 (4,091 MB), and 4.66% with H-PFD (4,850 MB). Notice that if we disregard the space usage of Interpolative encoding, the smallest space usage is achieved by S18 using IBDA order. The latter increases the space usage of Interpolative by 11.01%.

5.3.2. 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 and URL order, S9 (the best existing trade-off from [41]) achieves 5.53 milliseconds per query. Using IBDA order and our H-PFD approach, we obtain 4.84 milliseconds per query (an improvement of 12.11%). If, alternatively, we use S18, we obtain 5.14 milliseconds per query (an improvement of 7.05%).

Notice also the following: if we consider S9 and S18, both on URL order, query times change slightly: from 5.53 to 5.52 milliseconds per query, respectively. If, on the other hand, we consider using IBDA instead of URL order, the improvement is bigger. This shows again the effectiveness of IBDA, as it is able to improve the already highly-efficient query times of AND queries [41].

5.3.3. WAND Queries

We also test with the WAND query-processing algorithm [11]. We use tf-idf ranking and top-10 results. As it can be seen in Table 4, we are able to speed-up the query processing not only by using IBDA, but also our hybrid compression methods.

For instance, for IBDA and using S18, we can improve query time from 59.50 milliseconds per query (S9 using URL order) to 47.10 milliseconds per query (an improvement of about 20.84%). For H-PFD using IBDA order, the improvement is of about 23.44% (compared to OptPFD using URL order).

Table 5 shows the average number of docIDs (in millions) that are decoded per query (recall that runs are regarded as single integers) and average number of blocks (in thousands) that are decompressed per query.

Our results indicate that the number of docIDs decoded (as well as the number of decompressed blocks) is reduced in two ways. First, by using one of our hybrid compression schemes. Second, by using

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

Query Algorithm	Compression Scheme	Reorder Method		
		`Unordered`	`URL`	`IBDA`
AND	VByte	42.85	22.88	28.88
	S9	11.97	5.53	5.43
	OptPFD	11.90	5.57	5.56
	H-VByte	19.73	7.49	6.91
	S18	12.67	5.52	5.14
	H-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
	H-VByte	184.74	66.95	54.57
	S18	150.87	59.14	47.10
	H-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
	H-VByte	340.29	83.99	41.62
	S18	475.70	141.58	59.97
	H-PFD	465.35	177.10	66.03

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

Compression Scheme	Reorder Method					
	`Unordered`		`URL`		`IBDA`	
	docIDs	Blocks	docIDs	Blocks	docIDs	Blocks
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
H-VByte	6.93	60.53	1.71	15.51	1.30	11.58
S18	10.89	81.85	3.21	24.21	2.09	15.87
H-PFD	10.91	85.44	3.60	31.90	2.22	18.24

`IBDA` rather than `URL` order. For instance, for `URL` order, if we use S18 instead of S9, we reduce by 29.89% the average number of blocks accessed at query time. If, besides, we now use `IBDA` order with S18, we obtain a further reduction of 34.45% compared to S18 with `URL` order. Overall, by using `IBDA` order and then S18 compression, rather than `URL` order and S9 compression [41], we reduce by 54.04% the average number of blocks accessed per query.

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 seen in Table 4) but also in cases where accessing blocks is expensive: think, for instance, of disk-resident inverted lists, whose blocks need to be transferred from secondary storage.

Finally, we think that this query-time improvement can be also achieved by the Block-Max WAND algorithm from [31].

5.3.4. Full OR Queries

We study now the performance of OR queries. In particular, full OR queries, where the full result (rather than a ranking) must be obtained. Assume that we need to compute OR for two inverted lists I_1 and I_2 (the following process can be extended to more than two 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 $d' + \ell' + 1$. 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 $d'' + \ell'' + 1$. We repeat this process until the current docID d^* we are looking for is not within a run. In this case, we can report the interval $[d', d^* - 1]$. This allows us to compute the union of these intervals without decompressing them.

Table 4 shows the experimental results. For VByte, S9, and PForDelta we use the traditional DAAT OR processing. As it can be seen, our algorithm for processing OR queries introduces important improvements. In particular, for H-VByte on IBDA, where the query time is reduced by 85.75% compared to VByte on URL. The rationale behind this exceptional performance of H-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. Notice also that IBDA is key for yielding the important improvements achieved by our hybrid compression approaches.

5.3.5. Comparison with Partitioned Elias-Fano Indexes

The *Partitioned Elias-Fano* encoding [26] has shown to be effective for ordered document collections. In particular, this method is mainly optimized for space usage. According to the results from Ottaviano and Venturini [26] (and validated by our own experiments), Partitioned Elias-Fano can be between 8% to 9% smaller than our hybrid schemes. For query processing, although being competitive, the results from Ottaviano and Venturini indicate that their approach outperforms gap-encoded indexes only for AND queries. In particular, when compared with OptPFD on URL order, they improve AND query processing time by about 2.17% to 19.29% [26], for ranked and boolean queries, respectively. In our case, the improvement compared with OptPFD on URL order ranges between 4.12% (H-PFD on URL order) to 4.84% (H-PFD on IBDA order). It is well-known, however, that Elias-Fano indexes are faster for AND queries, in particular for selective queries [37, 26].

For boolean OR and WAND queries, on the other hand, Partitioned Elias-Fano indexes are slower than gap-encoded indexes [26]. Our hybrid compression approaches, on the other hand, excel in these cases, taking advantage of two facts to improve query processing time:

1. The implicit decompression of runs, which is key for boolean OR queries. Elias-Fano indexes, on the other hand, still need to do explicit decompression of consecutive docIDs.
2. The reduction in the number of inverted-list blocks accessed (and decompressed), which is key for WAND queries.

Overall, this shows that our hybrid approaches offer an interesting space/time trade-off, compared to Partitioned Elias-Fano indexes.

5.4. Application to Inverted List Caching

Finally, we test our compression approaches for increasing amounts of inverted lists. In our experiments, we use as input the TREC GOV2 inverted index with docIDs assigned by URL order. 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 to least frequent). Next, we consider the top-Q lists in this sorting, for $Q = 1,000, \ldots, 10,000$. For each such Q, we apply IBDA on the top-Q lists, using the order we already mentioned.

Table 6 shows the experimental results. We include just S18 compression since it yields the smallest space usage among the tested schemes. As it can be seen, S18 yields a considerable improvement in space usage, either on URL and IBDA sorting. For instance, we can conclude that 10,000 lists compressed with S18 on IBDA order use about the same amount of space than 2,000 lists compressed with S9 on URL. In other words, within the same space used by S9 on URL to achieve a 24% hit ratio, S18 on IBDA is able to achieve a 51% hit ratio.

Finally, notice how the improvement of space usage achieved by S18 IBDA (compared to S9 URL) degrades as we compress more lists. For 1,000 lists and using S18 IBDA, the improvement of space usage is 30.10%. For 10,000 lists, on the other hand, the improvement is 22.54%. Moreover, recall that for the whole index we obtain a reduction of about 10%, recall Tables 1 and 3.

6. Conclusions and Future Work

Our main conclusion is that for ordered document collections, hybrid run-length + gap encoded inverted indexes outperform traditional gap encoded inverted indexes. This is supported by the following results:

Table 6: Space usage (in MB) for the most frequently queried inverted lists.

Top-Q lists	S9 URL	S9 IBDA	S18 URL	S18 IBDA	Improvement	Hit Ratio
1,000	1,011	958	772	699	30.10%	13%
2,000	1,329	1,266	1,042	958	27.91%	24%
3,000	1,509	1,443	1,211	1,122	25.65%	31%
4,000	1,610	1,543	1,307	1,218	24.35%	37%
5,000	1,670	1,603	1,365	1,276	23.59%	40%
6,000	1,704	1,639	1,399	1,311	23.06%	42%
7,000	1,723	1,659	1,417	1,331	22.75%	44%
8,000	1,733	1,669	1,427	1,341	22.62%	46%
9,000	1,738	1,674	1,431	1,346	22.55%	49%
10,000	1,739	1,675	1,432	1,347	22.54%	51%

- A reduction of space usage of about 10% (on average) for the whole inverted index, compared to the most efficient gap-encoded alternatives from Yan et al. [41].
- A decompression speed that is up to 4.58 times faster than the most efficient gap-encoded alternatives from Yan et al. [41].
- An improved DAAT query processing time of AND (by up to 12%), WAND (by up to 23%), and full OR queries (by up to 86%), compared to the most efficient gap-encoded alternatives from Yan et al. [41].
- A reduction of up to 55% in the number of inverted-list blocks that are accessed (and decompressed) at query time. This is important for disk-resident inverted lists, as the number of blocks that must be transferred to main memory is reduced.

These results are obtained by using the hybrid compression approaches introduced in this paper, along with the document reordering method introduced to generate long runs of consecutive docIDs in inverted lists. Our experiments also indicate that our reordering method is useful to improve the decompression speed of the traditional gap-encoding approaches [41]. When compared with a document collection ordered by URLs, we obtain a decompression speed that is 1.22 faster.

As future work, we plan to test our approaches to improve the performance of the Block-Max WAND algorithm of [31]. It would be also interesting to use the approach of [34] along with ours. A list of interesting lines for future work is as follows:

- It would be interesting to take advantage of the runs to improve query processing time. These can be seen as precomputed intervals, which can be intersected more efficiently using ideas like interval algebra [1, 13].
- We showed that being able to generate runs in inverted lists is advantageous for our hybrid compression approaches. However, a more in-depth study is needed to see how the existing reordering methods generate runs.
- It would be interesting to test how our hybrid compression approaches behave in the scenario of versioned document collections [19, 14]. In these cases, having runs of consecutive docIDs is a natural feature.
- It would be interesting to know whether our hybrid compression approaches can be adapted to compress bitmap indexes in databases [40]. One of the most efficient alternatives in practice is *Word Aligned Hybrid* (WAH), which combines run-length encoding with other compression approaches. As we have seen in this paper, we are able to reduce the number of accessed blocks significantly, which would help to reduce the accesses to secondary storage.
- Finally, it would be interesting to test our hybrid approaches for graph compression [9].

References

[1] Allen, J. F., 1981. An interval-based representation of temporal knowledge. In: Proc of 7th International Joint Conference on Artificial Intelligence (IJCAI). pp. 221–226.

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

[3] Anh, V. N., Moffat, A., 2006. Improved word-aligned binary compression for text indexing. IEEE Transactions on Knowledge and Data Engineering 18 (6), 857–861.

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

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

[6] Barla Cambazoglu, B., Baeza-Yates, R., 2015. Scalability Challenges in Web Search Engines. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers.

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

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

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

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

[11] Broder, A., Carmel, D., Herscovici, M., Soffer, A., Zien, J., 2003. Efficient query evaluation using a two-level retrieval process. In: Proc. of 12nd Int. Conference on Information and Knowledge Management (CIKM). ACM, pp. 426–434.

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

[13] Clarke, C. L. A., Cormack, G. V., Burkowski, F. J., 1995. An algebra for structured text search and a framework for its implementation. The Computer Journal 38 (1), 43–56.

[14] Claude, F., Fariña, A., Martínez-Prieto, M. A., Navarro, G., 2016. Universal indexes for highly repetitive document collections. Information Systems 61, 1–23.

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

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

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

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

[19] He, J., Zeng, J., Suel, T., 2010. Improved index compression techniques for versioned document collections. In: Proc. of 19th ACM Conf. on Information and Knowledge Management (CIKM). ACM, pp. 1239–1248.

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

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

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

[23] Lemire, D., Kaser, O., Aouiche, K., 2010. Sorting improves word-aligned bitmap indexes. Data and Knowledge Engineering 69 (1), 3–28.

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

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

[26] Ottaviano, G., Venturini, R., 2014. Partitioned elias-fano indexes. In: Proc. of 37th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 273–282.

[27] Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Lioma, C., 2006. Terrier: A high performance and scalable information retrieval platform. In: Proc of ACM SIGIR Workshop on Open Source Information Retrieval (OSIR).

[28] Ramaswamy, V., Konow, R., Trotman, A., Degenhardt, J., Whyte, N., 2017. Document reordering is good, especially for e-Commerce. In: Proc of ACM SIGIR Workshop on eCommerce (eCom). 6 pages.

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

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

[31] Shuai, D., Suel, T., 2011. Faster top-k document retrieval using block-max indexes. In: Proc. of 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 993–1002.

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

[33] Silvestri, F., Orlando, S., Perego, R., 2004. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In: Proc. of 27th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 305–312.

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

[35] Tonellotto, N., Macdonald, C., Ounis, I., 2011. Effect of different docid orderings on dynamic pruning retrieval

strategies. In: Proc. of 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 1179–1180.
[36] Turpin, A., Tsegay, Y., Hawking, D., Williams, H., 2007. Fast generation of result snippets in web search. In: Proc. of 30th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 127–134.
[37] Vigna, S., 2013. Quasi-succinct indices. In: Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM). pp. 83–92.
[38] Williams, H., Zobel, J., 1999. Compressing integers for fast file access. The Computer Journal 42 (3), 193–201.
[39] Witten, I., Moffat, A., Bell, T., 1999. Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition. Morgan Kaufmann.
[40] Wu, K., Otoo, E., Shoshani, A., 2006. Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems 31, 1–38.
[41] Yan, H., Ding, S., Suel, T., 2009. Inverted index compression and query processing with optimized document ordering. In: Proc. of 18th Int. Conference on World Wide Web (WWW). ACM, pp. 401–410.
[42] Zhang, J., Long, X., Suel, T., 2008. Performance of compressed inverted list caching in search engines. In: Proc. of 17th International Conference on World Wide Web (WWW). ACM, pp. 387–396.
[43] Zobel, J., Moffat, A., 2006. Inverted files for text search engines. ACM Comput. Surveys 38 (2).
[44] Zukowski, M., Héman, S., Nes, N., Boncz, P., 2006. Super-scalar RAM-CPU cache compression. In: Proc. of 22nd Int. Conference on Data Engineering (ICDE). IEEE Computer Society, p. 59.