Leveraging Context-Free Grammar for Efficient Inverted Index Compression

Zhaohua Zhang[1] Jiancong Tong[2] Haibing Huang[1] Jin Liang[2]
Tianlong Li[2] Rebecca J. Stones[1] Gang Wang[1]† Xiaoguang Liu[1]†

[1]College of Computer and Control Engineering & College of Software, Nankai University, China
[2]Baidu, Inc.
{zhangzhaohua, jctong, hbhuang, liangjin, litianlong, becky, wgzwp, liuxg}@nbjl.nankai.edu.cn

ABSTRACT

Large-scale search engines need to answer thousands of queries per second over billions of documents, which is typically done by querying a large inverted index. Many highly optimized integer encoding techniques are applied to compress the inverted index and reduce the query processing time. In this paper, we propose a new grammar-based inverted index compression scheme, which can improve the performance of both index compression and query processing.

Our approach identifies patterns (common subsequences of docIDs) among different posting lists and generates a context-free grammar to succinctly represent the inverted index. To further optimize the compression performance, we carefully redesign the index structure. Experiments show a reduction up to 8.8% in space usage while decompression is up to 14% faster.

We also design an efficient list intersection algorithm which utilizes the proposed grammar-based inverted index. We show that our scheme can be combined with common docID reassignment methods and encoding techniques, and yields about 14% to 27% higher throughput for AND queries by utilizing multiple threads.

Keywords

Inverted index compression, context-free grammar, query processing

1. INTRODUCTION

The most widely used data structure in current search engines is the *inverted index*, which allows the operator to find documents that contain particular terms efficiently [34]. The inverted index of a commercial search engine typically occupies a large fraction of total storage, so the index is ordinarily compressed. A smaller index not only means less space is needed but also decreased transmission time between the disk and main memory.

†Corresponding authors.

SIGIR '16, July 17-21, 2016, Pisa, Italy
© 2016 ACM. ISBN 978-1-4503-4069-4/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2911451.2911518

Previous work on index compression techniques [1,2,37] mainly focus on various integer encoding schemes that aim to compress the identifiers of documents (docIDs, which are integers) in the inverted index better. Since these techniques are often concerned with compressing integer sequences whose values are small on average, their resulting compression ratios depend heavily on the way in which docIDs are assigned [6,28,35].

This paper is instead dedicated to a new compression scheme which improves the compression performance by removing the duplicate data before encoding. We note that duplicate data is dominant in the inverted index, with different posting lists containing common subsequences of docIDs, which we call *patterns*. We devise an algorithm called *PISEQUENTIAL* (Pattern Identification Sequentially) to identify patterns in the inverted index. Like other applications of the grammar-based method in compressing non-text data, it is difficult to obtain satisfactory compression ratio and decompression speed if we encode the generated grammar directly. To improve compression performance, we design a partitioned index structure. Besides compressing the inverted index, common patterns can also be utilized to improve the efficiency of query processing by eliminating unnecessary docID comparison operations. For AND queries, we propose an efficient intersection algorithm for the grammar-based inverted index, employing document reordering methods and run-length encoding. Moreover, the proposed index structure also supports OR and WAND [7] query processing.

2. PRELIMINARIES

2.1 Inverted Index and Compression

The inverted index is a simple yet powerful data structure used in search engines. Given a collection of N documents, each document will be identified by a unique docID from 1 to N. An inverted index consists of many posting lists. Each posting list corresponds to a unique term and contains all documents where this term occurs in the collection. For a term t, the posting list $l(t)$ typically has the structure

$$\langle d_1, d_2, d_3, \ldots, d_{f_t} \rangle$$

where f_t is the number of documents that contain t, and d_i is the docID of the i-th document containing t. Since the docIDs of a posting list are generally stored in an ascending order, modern search engines usually take differences between adjacent docIDs to convert $l(t)$ into a sequence of d-gaps $l'(t)$

$$\langle d_1, d_2 - d_1, d_3 - d_2, \ldots, d_{f_t} - d_{f_t - 1} \rangle$$

and then compress $l'(t)$ instead of $l(t)$.

To skip unnecessary subregions when processing queries, posting lists are often split into blocks of, say, 128 d-gaps each, so that only the blocks that are relevant to a query need to be accessed. Although we have to store a mapping table for fast block locating, the extra space occupied by it is much smaller than that used by the inverted index itself.

In the context of a search engine, inverted index compression (encoding) is usually infrequent compared to decompression (decoding), which must be performed for every uncached query. As grammar-based compressors often need a high amount of memory and time to run, this paper not only focuses on decompression performance but also compression performance. We not only consider the compression and decompression algorithms of the docIDs, but also how to store other pertinent information needed by processing scored queries in grammar-based index, such as term frequency.

2.2 Query Processing

For a given query, which is interpreted as a set of terms, the common query operations are *conjunctive queries* and *disjunctive queries*. Conjunctive (AND) queries are used to identify the subset of documents which contain all search terms. Disjunctive (OR) queries retrieve the documents which contain at least one term in the query. In most scenarios, each result document of one query should be associated with a *relevance score*. The score indicates the relevance between the query and the document. Most search engines use the *k* highest scored documents as the final retrieval result. One of the most popular relevance ranking methods is BM25 [26], which is used in our experiments.

To illustrate, assume the query “2016 Summer Olympics" is made. The search engine will find the posting lists l_1, l_2 and l_3 for the three terms “2016", “Summer" and “Olympics", respectively, which may look like

$$l_1 = \langle 1, 2, 3, 14, 20, 21, 39, 40, 49, 51, 55\rangle,$$
$$l_2 = \langle 1, 2, 3, 9, 10, 11, 14, 21, 39, 40, 49, 55\rangle \text{ and}$$
$$l_3 = \langle 1, 2, 3, 14, 16, 39, 49, 53, 55\rangle.$$

If the query above is processed as one conjunctive query, the intersection algorithm returns

$$l_1 \cap l_2 \cap l_3 = \langle 1, 2, 3, 14, 39, 49, 55\rangle$$

and the disjunctive algorithm returns

$$l_1 \cup l_2 \cup l_3 = \langle 1, 2, 3, 9, 10, 11, 14, 16, 20, 21, 39, 40, 49, 51, 53, 55\rangle.$$

Two basic techniques for traversing posting lists are *Document-At-A-Time* (DAAT) and *Term-At-A-Time* (TAAT) [8]. DAAT maintains a pointer to the “current" posting for each list, and moves the pointers forward in parallel as the query is being processed. TAAT traverses posting lists one by one, and uses a temporary data structure to keep track of the current candidates. In our experiments, we use TAAT for Boolean conjunctive queries and disjunctive queries, while for WAND queries we use DAAT.

2.3 Grammar-based Compression

A *context-free grammar G* is a quadruple (V_T, V_N, S, P) where V_T is a finite *alphabet* whose elements are called *terminals*, V_N is the set of *non-terminals*, and S is a special non-terminal called the *start symbol*. In general, the word *symbol* refers to any terminal or non-terminal. The last component P is a set of *production rules* of the form $A \to \alpha$, where A is a non-terminal and α is a string of symbols referred to as the *definition* of A.

A production rule $A \to \alpha$ can be rewritten to a string u, by iteratively replacing each non-terminal by its definition until only terminals remain. The set of strings derived from the start symbol is denoted by $L(G)$. A grammar G is *admissible* if $|L(G)| = 1$ and for each non-terminal A, there is exactly one production rule $A \to \alpha$ defined in P. Since we use a grammar G to represent and compress a unique inverted index, this paper only considers admissible grammars. Define $|G|$ as the total length of strings on the right hand sides of all production rules.

The central idea behind grammar-based compression is to use a context-free grammar to represent the input and reduce repeating patterns captured. For example, the string “$abaababa$” could be represented by the grammar

$$(\{a, b\}, \{Q, L, S\}, S, \{Q \to aba, L \to Qab, S \to LQ\}).$$

After inferring a grammar that represents the input string, these methods [17, 18, 21, 36] often convert the grammar to a symbol stream, and then transform it into a bit stream by an entropy encoder, which affects the final size of the compressed file. A decoder simply proceeds backwards.

3. RELATED WORK

3.1 Inverted Index Compression

Inverted index compression aims not only to reduce the space consumption of index files, but also to support efficient query processing. As posting lists in the inverted index are usually represented as strictly increasing sequences of integers (i.e., docIDs) together with term’s frequency, search engines utilize mathematical encoding methods to compress these lists. Since many mathematical encoding methods aim to use fewer bytes (or bits) to represent the strictly monotone sequences, *delta encoding* has been widely used to achieve a high level of compression. Most previous work on index compression usually assumes that the posting lists are turned into d-gap lists and mainly focuses on encoding method.

Since topics related to compressing integer sequences have been studied several decades, many solutions have been proposed for different trade-offs between compression ratio and decompression speed, while both aspects are important for inverted index compression. In this paper, however, we just limit our discussion to several established integer encoding techniques suitable for this problem. VByte [33] encodes an integer using a variable number of bytes, where each byte consists of one status bit and 7 data bits. The status bit indicates whether the current byte is the last one in the representation of the integer or not. VByte does not achieve a good compression ratio, because it is unsuitable for compressing small integers. But it allows for fast decoding as the encoding is byte-aligned, and is thus used in many systems. Stepanov et al. [32] present a variant coding method based on VByte (called Varint-G8IU) which exploits SIMD instructions in modern CPU for faster decoding.

In addition, PForDelta [38] also aims at fast decompression. It divides the list of integers into segments of length s, always divisible by 32. To encode the integers within a given segment a, it first determines the smallest b such that most integers in a (say, 90%) are less than 2^b and thus can be stored using b bits. The remaining values, called *exceptions*, are encoded separately. In the slot of each exception, it maintains a pointer to the location of the next exception, forming a linked list. One popular variant of PForDelta is OptPFD, which is introduced in [35]. Instead of setting a constant threshold of the number of exceptions per block, OptPFD makes the selection an optimization problem to achieve the best trade-off between compression ratio and decoding speed.

Simple16 (S16) is also a widely used algorithm proposed in [37] that achieves both good compression and high decompression speed.

Compared with Simple9 (S9) [2], since each case of S16 uses all data bits, it achieves slightly better compression.

As most integer encoding methods require sequential decoding, each posting list is split into blocks, where each block can be encoded and decoded independently. In most cases, the size of block is fixed, e.g. 128 or 256 integers [10, 20]. Silvestri et al. [31] introduce an optimal partition strategy for partitioning an integer sequence into blocks to get better compression ratio.

In addition to mathematical encoding methods, there are other approaches for index compression. Beskales et al. [5] propose an approach to map terms in a document collection to a new term space, thereby to generate a more compact inverted index for better compression. Giuseppe et al. [24] describe an index scheme based on dividing posting list into chunks and compressing them with Elias-Fano code, thereby forming a two-level index structure. Their method takes advantage of the local statistics of the chunk for better encoding, thus improving compression. Some works also focus on compressed indexes used in labeled graphs [14], or combined with phrase-based ranking [25], and space-time tradeoff [23].

3.2 Document Reordering

To improve inverted index compression, a number of document reordering (*docID reassignment*) methods have been developed. The key idea of this kind of approach is to actively enhance the clustering property of posting lists so that similar documents have close docIDs, thereby improving the performance of integer encoders. The approaches proposed in [6, 30] use graph structures to represent the relationship among documents and assign docIDs during a graph traversal. A much simpler yet effective approach was proposed in [29], which assigns docIDs alphabetically according to their URLs.

Recently, Arroyuelo et al. [3] proposed a reordering method based on run-length encoding which is able to create longer runs of d-gaps of 1 (or just 1s, for short). By representing each run with just two values when encoding the posting lists, they showed that space usage can be reduced. Shi et al. [27] instead achieved longer runs of 1s by reordering according to the document frequencies of terms in the inverted index. These two methods not only enhance the clustering property, but also provide more runs, thereby achieving better compression than run-length encoding. Moreover, for every occurrence of a specific run with length l, only two instead of l values need to be stored no matter how large l is, so the two methods actually reduce redundant storage for the multiple occurrences of a common docID sequence.

In this paper, we present a more general and powerful pattern identification algorithm which improves common encoding techniques and reordering methods.

3.3 List Intersection

Sorted lists intersection has been studied for several decades. There are many searching methods proposed such as linear search, interpolation search [15] and galloping search [4]. However, as the inverted lists are typically stored compactly by using the methods mentioned in Section 3.1, some special algorithms are proposed to accelerate intersection on these compact sequences. Gupta et al. [16] achieved good asymptotic performance by introducing a two-level data structure in which each level is itself searchable and compressed. Culpepper et al. [13] proposed a simpler hybrid method that can provide both compact storage and faster lists intersection. However, as described in Section 2.1, these methods need to store an auxiliary index, and decompression will be much slower than intersection when processing queries. In [3], a novel lists intersection algorithm was provided which reduces explicit decompression by directly performing range checking on the runs. Although it still stores auxiliary index, as fewer docIDs need to be decoded, it reduces the query processing time effectively.

Compared with these algorithms, the scheme presented in this paper aims at: utilizing a grammar-based index structure to reduce the redundant comparison operations, and exploiting reordering methods and encoding techniques such as run-length encoding to reduce extra overhead, e.g., random memory access overhead.

3.4 Grammar-based Compression

Grammar-based compression is an active research area with a wide variety of applications. Nevill-Manning and Witten [21, 22] proposed an on-line linear-time algorithm, called *Sequitur*, which infers a context-free grammar to losslessly represent the input. Yang and Kieffer [36] improved Sequitur to make it universal. *Re-Pair* [18], proposed by Larsson and Moffat, is an off-line algorithm that infers a dictionary by recursively creating the phrases that occur most frequently. Although these methods have shown success for many different types of data (in fact, some of them are known to be asymptotically optimal on input strings generated by finite-state sources), Charikar et al. [9] showed that many of the best-known compressors can fail dramatically. A good grammar-based compression algorithm often attempts to find the smallest possible grammar generating the input string, but a smaller grammar does not necessarily mean a smaller compression ratio as described in [36], as many practical issues are ignored.

Recently, Claude et al. [11] introduced a new compressed inverted index by using grammar-based compression for highly repetitive document collections. They use *Re-Pair* as their grammar compressor and then compress the sequence formed by concatenating all of the d-gap lists. They also add extra information to non-terminals that enables fast skipping over the compressed lists without decompressing. According to their experiments, their methods significantly reduce the space achieved by classical compression, at the price of moderated slowdowns on word and conjunctive queries. Moreover, they discuss some possible extensions in [12], such as supporting ranking capabilities within their index. Instead of the particular case of highly repetitive collections, we focus on a more general one, in which there may not be enough repetition to make up for the expansion that using a grammar causes. Therefore, in addition to making/finding more common patterns among posting lists, another important task of this paper is to reduce the overhead by using some auxiliary methods.

4. THE PROPOSED SCHEME

The key idea of the proposed compression scheme is to identify and remove repeating patterns among posting lists before encoding them. In Section 4.1, we provide the pattern identification algorithm. To illustrate, suppose the posting lists l_1, l_2, l_3 described in Section 2.2 need to be compressed. After processed by our algorithm, they are represented as a grammar G that consists of the production rules:

$$\begin{aligned} &A \to (1, 2, 3) \\ &B \to (21, 39, 40, 49) \\ &{l_1}' \to (A, 14, 20, B, 51, 55) \\ &{l_2}' \to (A, 9, 10, 11, 14, B, 55) \\ &{l_3}' \to (A, 14, 16, 39, 49, 53, 55) \end{aligned}$$

where A and B are the identifiers of the *patterns* found in l_1, l_2, l_3, and ${l_1}', {l_2}', {l_3}'$ are the *reduced posting lists*. The identifiers of patterns and reduced posting lists together make up the set of non-terminals. For this example, the non-terminals are $\{A, B, l_1', l_2', l_3'\}$.

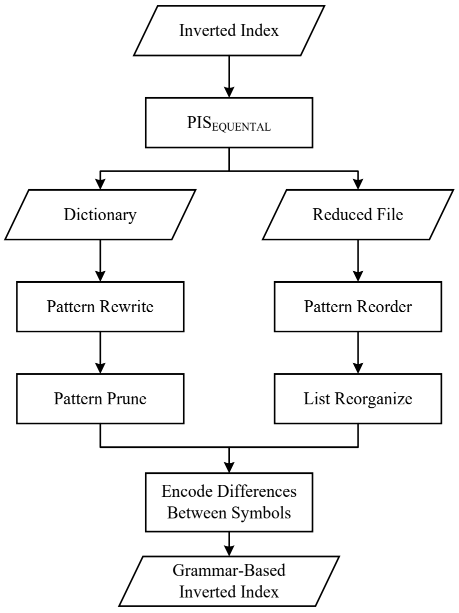

Figure 1: Flowchart of the proposed Grammar-based Compression Scheme

The remaining docIDs are terminals, which, for the above example, are $\{1, 2, 3, 9, 10, 14, \ldots, 55\}$. Since we must express distinct lists rather than a single input, this grammar has a set of start symbols rather than a single start symbol (in the example above, $\{l_1{}', l_2{}', l_3{}'\}$). So we represent the grammar by a quadruple $G = (DI, PI \cup RI, RI, PP \cup RP)$, where DI, PI and RI respectively denote the identifier sets of documents, patterns and reduced lists, and PP and RP denote the pattern and reduced list production sets respectively. Since the repetitions are removed, the reduced index size $|G|$ is smaller than that of the original index size $|l_1| + |l_2| + |l_3|$. The complete compression scheme is shown in Figure 1.

In our implementation, we have found that the overall memory usage for grammar generation is a critical problem. For the 12GB original index file we use, the process of pattern identification would cost more than 64GB memory. To reduce memory usage, we use a hash segmentation strategy to partition the original index into segments, where the grammar generation on each segment is independent from others. In Section 4.2, we will give the details of the index partition. Moreover, the generated grammar is unsuitable for encoding directly, so we propose some approaches to improve the compression performance. Section 4.3 introduces how we reorganize the generated grammar.

4.1 Pattern Identification

The key to grammar-based compression is to effectively find and remove repetitions occurring in the data. In this subsection, we modify a widely used grammar-based compression algorithm [36], *Sequential*, to work on an inverted index, which we call PISequential.

The details of PISequential are shown in Algorithm 1. The function find_longest_prefix(l_i, j, PP) finds the longest prefix of the unprocessed portion of l_i beginning with $l_i[j]$ that matches the expansion of some pattern production rule in PP. If found, it returns the pattern identifier; otherwise, $l_i[j]$ is returned. In Algorithm 1, freq(p) denotes the number of times p occurs in the right-hand side of production rules in $PP \cup RP$.

Algorithm 1 PISequential for pattern identification

```
Input: Inverted index $I$ including posting lists $\{l_1, l_2, \ldots, l_n\}$
Output: A grammar $G_0 = (DI, PI \cup RI, RI, PP \cup RP)$ that
        represents $I$
 1: Initialize $PI$, $RI$, $PP$ and $RP$ to $\emptyset$
 2: for each posting list $l_i$ of $I$ do
 3:     Create a new empty production rule $r_i \to \varepsilon$
 4:     for $j = 1$ to $|l_i|$ do        ▷ traverse the list from left to right
 5:         if $j = 1$ then
 6:             replace $r_i \to \varepsilon$ with $r_i \to l_i[1]$
 7:             $j := j + 1$
 8:             continue
 9:         end if
10:         $s :=$ find_longest_prefix$(l_i, j, PP)$
11:         $j := j + ||s||$
                ▷ $||s||$ is the length of pattern rule, or 1 if not found
12:         If currently $r_i \to \alpha$, set $b :=$ last_symbol$(\alpha)s$ and
            replace $r_i \to \alpha$ with $r_i \to \alpha s$
13:         if $b$ appears in $y$ in some $x \to y \in PP \cup RP$ then
14:             Create a new pattern $p \to b$
15:             In the production rules for $r_i$ and $x$, replace the oc-
                currences of the subsequence $b$ by $p$
16:             Add $p$ to $PI$ and $p \to b$ to $PP$
17:         end if
18:     end for
19:     Add $r_i$ to $RI$ and the production rule for $r_i$ to $RP$
20: end for
21: for each pattern $p \to \delta$ in $PP$ do
22:     if freq$(p) \times (|p| - 1) < |p| + 1$ then
23:         Replace all occurrences of $p$ in $PP$ and $RP$ by $\delta$
24:     end if
25: end for
26: Make identifiers in $RI$ consecutive and update productions in
    $PP$ and $RP$
```

The main difference compared with Sequential is that, PISequential processes posting lists one by one (lines 2-20) to find repeating patterns, which makes each posting list self-indexed, and separates the dictionary of found patterns from posting lists. Another important improvement happens after grammar generation. Since there's possibly upwards of millions of docIDs, most of the posting lists are relatively sparse. Therefore, the grammar-based method will naively generate a large number of short production rules, which do not combine into longer ones. In order to improve compression, PISequential subsequently removes some of these short production rules to improve the compression performance (lines 21-26).

After the grammar is generated, we store a dictionary containing the pattern productions PP and a file of the reduced posting list productions RP. To differentiate pattern identifiers from docIDs in posting lists, we set the most significant bit of a 32-bit word to represent a pattern identifier and use the reset bit to represent a docID.

4.2 Index Partition

In our implementation, we find that the corpora of bigrams (b on line 12 in Algorithm 1) temporarily uses at least 5.3 times more memory than the original index file, a significant constraint that also results in poor bigram search efficiency. The reason is that we need to keep a record of each possible bigram in the whole index and grammar list. Consequently, we partition the original inverted index into segments according to the most significant *K* bits of the docIDs. After this, we use Algorithm 1 to generate the grammar on each individual segment. Although some long patterns could be missed by this method, we reduce the overall memory usage when generating the context-free grammar since each segment is much smaller than the whole index. Moreover, as each segment is independent, the memory spent by the bigrams of previous segments can be reused. More details about index partition will be discussed in Section 5.2.

4.3 Grammar Reorganization

To save space, we store the gaps of docIDs instead of original values. Although there may be non-terminals in the list, we just skip them and turn each docID into the value by subtracting from the previous docID. If the preceding symbol is a non-terminal, the previous docID should be the last docID in the pattern. Take the grammar G described at the beginning of the Section 4 as an example, the grammar will be

$$\begin{aligned} &A \rightarrow (1,1,1) \\ &B \rightarrow (21,18,1,9) \\ &l_1{}' \rightarrow (A,11,6,B,2,4) \\ &l_2{}' \rightarrow (A,6,1,1,3,B,6) \\ &l_3{}' \rightarrow (A,11,2,23,10,4,2). \end{aligned}$$

Traditional integer encoding schemes are generally proposed on the assumption that the gaps are distributed according to a power-law distribution. Once the assumption does not hold, their compression performance may decline sharply. Therefore, we analyze the gap distribution of the generated grammar (G_0); see Figure 2.

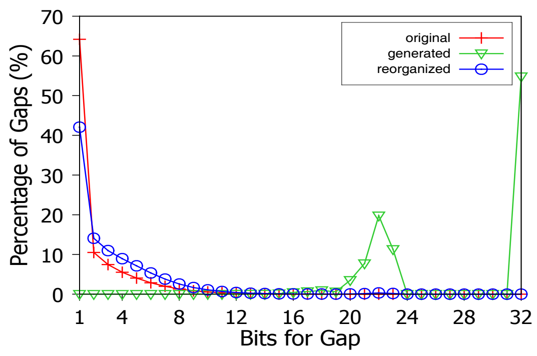

Figure 2: The proportion of gaps with a given number of bits for the *original lists*, the *generated grammar* (G_0), and the *reorganized grammar* (G_2) on the *GOV2* data set, of which docIDs are assigned by URL.

Figure 2 shows a decided difference between the gap distribution of the generated grammar (G_0 in Algorithm 1) and that of the original lists (which display a power-law distribution). As a result, it is difficult to obtain a satisfactory compression ratio by encoding G_0 directly. Moreover, random memory accesses caused by pattern fetching significantly impacts decompression speed, so we reorganize the grammar to address these issues.

Algorithm 2 rewrites each pattern production rule to a terminal string, i.e., it eliminates hierarchies in the grammar, with the aim of reducing the memory access overhead. It also removes all patterns whose length is smaller than a preset threshold L. Although this reorganization may result in larger space requirements, it avoids many random memory accesses, and so decompression and intersection are faster.

For the reduced posting list, since each pattern identifier is represented by a 32-bit integer, indicated by setting the most significant

Algorithm 2 Pattern rewriting and pruning

```
Input: The grammar $G_0 = (DI, PI \cup RI, RI, PP \cup RP)$ generated by PISEQUENTIAL and a threshold $L$
Output: A grammar $G_1$ in which the patterns are rewritten and the corresponding lists are updated
 1: for each pattern $p \rightarrow \alpha$ in $PP$ do
 2:     while $\alpha$ has non-terminal(s) do
 3:         Replace each non-terminal in $\alpha$ by its definition
 4:     end while
 5:     if $|\alpha| \leq L$ then    ▷ remove short patterns for performance
 6:         Replace each occurrence of $p$ by $\alpha$ in the right-hand side of any production rule in $PP \cup RP$
 7:         Remove $p \rightarrow \alpha$ from $PP$ and $p$ from $PI$
 8:     end if
 9: end for
10: Remove all unreferenced patterns from $PP$
```

bit to 1, patterns are effectively encoded as large integers, which will result in poor compression. Algorithm 3 eliminates the indicating bit, so the integers representing pattern identifiers are smaller, and can be stored in fewer bits. Because of this, all pattern identifiers in PP are reassigned in ascending order of their patterns (line 1 in Algorithm 3) and consequently the pattern identifiers in each reduced posting list remain in ascending order, which makes it possible to store their gaps rather than the pattern identifiers themselves. We convert each non-terminal (except the first one) in a production rule of a reduced posting list to its gap value, i.e., we subtract the previous non-terminal (line 4). We store these gaps (or non-terminal for the first element in the list) with one extra integer to indicate the offset to the next non-terminal. The extra offset for the last non-terminal of the list is zero. For each list, we also have to store one extra value to indicate the position of the first non-terminal.

Algorithm 3 Pattern reordering and list reorganizing

```
Input: The grammar $G_1 = (DI, PI \cup RI, RI, PP \cup RP)$ generated by Algorithm 2
Output: A grammar $G_2 = (DI, PI \cup RI, RI, PP' \cup RP')$ in which the pattern identifiers are reassigned and the reduced posting lists are reorganized
 1: Assign $\{1, 2, \ldots, |PP|\}$ to all pattern identifiers in $PP$ according to the ascending order of their patterns
 2: for each reduced posting list $l \rightarrow \beta$ in $RP$ do
 3:     Update non-terminals in $\beta$ using new pattern identifiers
 4:     Transform non-terminals in $\beta$ into gap lists
 5: end for
```

After these processes, the integers to be encoded are effectively decreased. In our running example, A and B will be respectively assigned the pattern identifiers 1 and 2 by Algorithm 3, and the production rules

$$\begin{aligned} &l_1{}' \rightarrow (A,11,6,B,2,4) \\ &l_2{}' \rightarrow (A,6,1,1,3,B,6) \\ &l_3{}' \rightarrow (A,11,2,23,10,4,2), \end{aligned}$$

will be encoded as

$$\begin{aligned} &l_1{}'' \rightarrow (1) \uplus ((1,3),11,6,(1,0),2,4) \\ &l_2{}'' \rightarrow (1) \uplus ((1,5),6,1,1,3,(1,0),6) \\ &l_3{}'' \rightarrow (1) \uplus ((1,0),11,2,23,10,4,2) \end{aligned}$$

where patterns are replaced by the pair

$$(\text{pattern identifier gap}, \text{distance to next pattern}).$$

The pattern identifier gap is defined as the pattern identifier minus the previous pattern identifier, except for the first pattern, when it is equal to the pattern identifier. The distance to next pattern is set to 0 for the last pattern. We also preappend the location of the first non-terminal, which happens to be 1 in all three cases in our example, which we denote (1).

The gap distribution in G_2 is comparable to the distribution for the original list, as indicated in Figure 2. Moreover, experiments indicate that the compression ratio can reach around 35% after transforming G_0 into G_2. It is thus possible to use mathematical encoding methods to compress reduced lists and patterns.

4.4 Query Processing on the Grammar-based Index

In this section, we describe how we conduct processing on the grammar-based index. We consider three query operations: AND queries, OR queries and WAND queries. Here we describe boolean AND queries, but the idea applies to the other operations.

In Section 4.4.1, we present the basic intersection algorithm suitable to our index structure. To produce more patterns among lists involved in the same query, Section 4.4.2 introduces several document reordering methods. To use patterns more effectively when processing queries, in Section 4.4.3, we restrict the patterns identified by PISEQUENTIAL to common "runs" among different posting lists rather than noncontinuous sequences.

4.4.1 Intersection Algorithm

The grammar-based index consists of not only the docIDs but also the patterns. Therefore, we design a new intersection algorithm suited to process queries on grammar-based indexes, shown in Algorithm 4. This algorithm has the same skeleton as the algorithms for plain inverted index; the key difference is the function INTERSECT which perform intersections on two reduced posting lists (lines 11 to 26 in Algorithm 4).

INTERSECT scans the two lists sequentially until one of them is exhausted. For each comparison between two elements, there are three possible scenarios: (1) If the two elements are identical, INTERSECT appends p directly to the resulting list, whether or not p and q are both non-terminals or terminals (docIDs). (2) If p and q overlap and both are non-terminals, it fetches their definitions and performs an ordinary document lists intersection. (3) Otherwise, if one element is docID d and the other is pattern, it fetches the pattern's production rule from the dictionary and searches for d within it, and then appends d to the result if found (lines 19 to 22). The algorithm finally rewrites the pattern list to a terminal string and merges it with the document list.

The first situation above saves many comparison operations by placing common patterns directly in the result (i.e. both elements are non-terminals). Profiling results show an advantage of Algorithm 4 over plain lists intersection in terms of the number of comparison operations required. However, many random memory accesses come as a result of fetching pattern definitions, offsetting this advantage. Consequently, we next explore how to make more patterns and use them more effectively.

We use a "skip" strategy to avoid some unnecessary random memory accesses. Specifically, before determining whether a docID exists in a pattern, we check the next element after the pattern first. For example, on line 20 in Algorithm 4 where p is the pattern and q is the docID, the next element n is $l_i[m+1]$. If n is also a docID and q is not smaller than n, we skip p and compare q with

Algorithm 4 List Intersection on a grammar-based index

```
Input: A $u$-term query $(t_1, t_2, \dots, t_u)$ and the compressed grammar $G$ that represents inverted index $I$
Output: The intersection results $\cap_{1\le i \le u} l(t_i)$
 1: Load the dictionary into the memory
 2: Re-label the $u$ lists so that $|l(t_1)| \le |l(t_2)| \le \cdots \le |l(t_u)|$
 3: $l = l(t_1)$                    ▷ $l$ keeps track of the current candidates
 4: $i := 2$
 5: while $i \le k$ do    ▷ intersect other lists $l(t_i)$ with $l$ one by one
 6:     $l' = $ INTERSECT$(l, l(t_i))$
 7:     $i := i + 1$
 8:     $l = l'$
 9: end while
10: return $l$

11: procedure INTERSECT$(l_i, l_j)$
12:     Create an empty set $r = \emptyset$
13:     for $m = 1$ to $|l_i|$ and $n = 1$ to $|l_j|$ do
14:         $p := l_i[m]$ and $q := l_j[n]$
15:         if $p = q$ then
16:             Add $p$ to $r$
17:         else if both $p$ and $q$ are non-terminals then
18:             Intersect production rules of $p$ and $q$, then add result(s) to $r$
19:         else if $p$ is a non-terminal then
20:             Search for $q$ in $p$, then add $q$ to $r$ if found
21:         else if $q$ is a non-terminal then
22:             Search for $p$ in $q$, then add $p$ to $r$ if found
23:         end if
24:     end for
25:     Return $r$
26: end procedure
```

elements after p. Otherwise, we will fetch the production rule for p and search for q in it. Experiments indicate this skip strategy can reduce random memory accesses by approximately 20%. Moreover, we utilize the prefetch CPU instruction to read production rules to hide the latency caused by cache misses, which achieves about a 13% speedup in AND query processing.

4.4.2 Pre-grammar Document Reordering

We test two existing document reordering methods and design a new method to produce more and longer common patterns among posting lists, especially lists involved in the same query.

Intersection-Based DocID Assignment (IBDA) [3] This aims to generate longer runs in the inverted index. For a given inverted index $L = \{l_1, l_2, \dots, l_n\}$, it computes $l_1 \cap l_2,\ l_1 \cap l_2 \cap l_3,\ \dots$, until $|l_1 \cap l_2 \cap \cdots \cap l_{j+1}| < M$, where M is a preset threshold. Then it assigns consecutive identifiers to the documents in $l_1 \cap \cdots \cap l_j$, and then to those in $l_1 \cap \cdots \cap l_{j-1} \setminus l_j$, and so on, until to $l_1 \cap l_2 \setminus l_3$. Then it removes the documents with reassigned docIDs from L, and repeats these steps until $L = \emptyset$. A related method was also described in [3], in which the co-occurrence of term pairs mined from the query log is used to determine the processing order of the lists, which we use for the experiments in this paper.

Term sorting-based(TRM) [27] This reordering method aims to produce smaller d-gaps. It first sorts posting lists in the descending order of their length, then sorts docIDs in the order in which they appear in the lists.

Frequency-Based Reordering (FBR) Grammar-based compression will benefit from highly repetitive sequences in the dataset, resulting in long duplicated sequences and the frequencies of them are higher. So we use a two-phase reordering method. After the index partition described in Section 4.2, we reorder the docIDs inside each segment in descending order of their frequency (within all posting lists in the same segment). Since the most frequent documents are moved to the head in each posting list, we then partition these docIDs into groups, with the number of docIDs are the same for each group (say 128 elements). In each group, we use the URL reordering method to assign final consecutive docID.

4.4.3 Run-length-based Intersection Algorithm

The document reordering methods result in more and longer patterns. However, pattern fetching introduces a large number of random memory accesses which significantly impacts query processing performance. To address this issue, we modify PISEQUENTIAL so that it only inserts new patterns that form a run. Since each run can be represented precisely by the minimum and the maximum docIDs, the output is significantly reduced. When we check whether one docID exists in the pattern, we only need to perform two comparison operations, i.e. to check if the docID is in the range of the pattern, and no more random memory accesses are required.

While this run-based grammar index structure may have a slight impact on the compression ratio, it improves query processing performance significantly. The proposed grammar-based scheme exploits run-length encoding like IBDA and TRM, but it employs a more general and powerful common pattern (run) identification algorithm.

5. EXPERIMENTS

5.1 Experimental Setup

To compare the various index schemes, we make use of full web documents from the *GOV2* collection as our document corpus, which results in an index of approximately 12 GB without positional information and frequencies. For scored queries, there is another index of about 12 GB storing frequency information and one file about 97MB storing document sizes. The docIDs are assigned according to the lexicographic order of their URLs [29]. We choose the *TREC 2009 Efficiency query set* (*T09*), which contains 32,244 queries, as our test query set.

We carried out all experiments on a PC server with a hexa-core 2.60 GHz Intel(R) Xeon(R) CPU and 64 GB of memory, running the CentOS 6.5. All algorithms are implemented in C++ and are compiled with g++ version 4.8.2, with optimization flag -O3. In our experiments, we use the IBDA, TRM and FBR document reordering methods. We implement IBDA and TRM based on their descriptions in their corresponding papers. For the integer encoders, we use the highly-efficient implementations from [19]. In our implementation, we use OptPFD to compress the dictionary, the reduced file and term frequency information. We encode each reduced list in blocks of 128 elements (docID or non-terminal) and also store a separate array about the maximum docID of each block for fast skipping during list intersection. The term frequencies of postings are stored apart from grammar index and kept the same order to posting lists. For each non-terminal in the reduced list, we also store the offset to the next frequency in the term frquency list, since there are more than one posting in the pattern.

We make use of the state-of-art indexes from [24] as our baselines. We also use the code made available by the authors of [11] to compare our method with RePair-Skip. We find that RePair-Skip get worse results than our methods and baselines. Due to limited space, we ignore the results of RePair-Skip here. To validate grammar-based indexes against the partitioned Elias-Fano and block-based indexes, we compared the construction consumption during index building, compression ratio and query process time between our methods and baselines. We refer to the grammar-based index, Elias-Fano index and block-based index as GM, EF and BLK, respectively. GM is the index generated by PISEQUENTIAL, while GM-Run is a modified version of GM which only identifies the patterns which form a run. At the beginning of experiments of decompression and query processing, all indexes are loaded into the memory.

5.2 Memory Usage in Pattern Identification

Before comparing the performance of the different indexes, we first investigate the memory space spent on pattern identification, of which the cost is larger than other procedures when compressing the inverted index with our methods.

As described in Section 4.2, we find that memory usage for pattern identification is temporarily at least 5.3 times more than original index file. Consequently, we partition the inverted index before grammar generation. According to the most significant K bits of the docID, we split each posting list into 2^K parts (some parts may be empty), and PISEQUENTIAL will be executed on each part across different lists individually. The memory usage and space usage as K varies is shown in Figure 3.

As we would expect, as K increases, the peak value of memory usage for each segment (solid lines) becomes smaller as fewer temporary data structures need to be maintained by PISEQUENTIAL. The peak memory usage is approximately 11.5 GB for GM and 12.8 GB for GM-Run when K is 3. When K is less than 3, the memory consumption for temporary data structures exceeds 64GB, which is the physical memory capacity in our server.

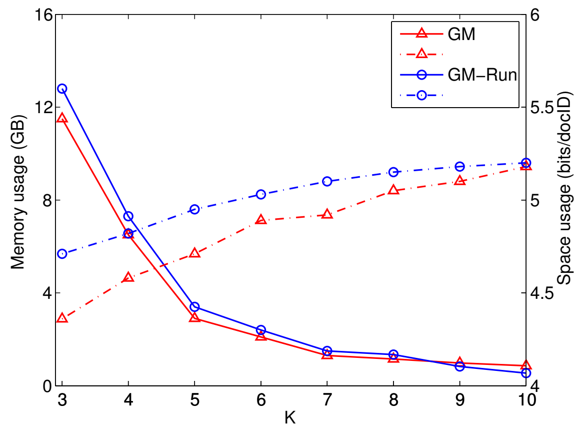

Figure 3: Peak memory usage (solid) and space usage (dashed) of GM and GM-Run.

We also find that the space usage (dashed lines in Figure 3) grows with K on both GM and GM-Run, implying the proposed method remains functional for larger indexes. As the value of K increases, more long candidate patterns have been cut off since the origin posting list becomes shorter for each segment. Based on these observation, we select $K = 3$ for our grammar-based index.

5.3 Compression and Decompression

5.3.1 Space usage

In Table 1 we list the compression ratio in two different formats: (a) the average number of bits required for each docID in the compressed lists, and (b) the overall space in gigabytes (without frequencies).

Table 1: Space usage of the index schemes for different document reordering methods

Index Scheme	Space (bits/int)				Space (GB)			
	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM
EF Uniform	5.41	4.98	6.13	5.83	2.19	1.89	2.29	2.21
EF ϵ-optimal	4.77	4.38	5.98	5.68	1.83	1.66	2.24	2.13
BLK OptPFD	5.21	4.63	6.43	5.93	1.98	1.74	2.43	2.25
BLK VByte	10.32	9.83	11.24	10.20	4.12	3.81	4.47	4.01
GM	4.35	4.01	5.82	5.31	1.78	1.58	2.20	1.99
GM-Run	4.41	4.33	5.97	5.61	1.79	1.64	2.23	2.12

We make the following observations:

- Both GM and GM-Run outperform the baselines on compression. The improvement over the best baseline EF ϵ-optimal is up to 8.8%.
- The compression ratio of GM is slightly better than GM-Run. This is because GM-Run only contains the run patterns in its dictionary while some other patterns are not extracted from the inverted index.
- All schemes benefited from the use of the FBR reordering method, and the best compression ratio overall is achieved by GM with FBR reordering.

5.3.2 Decompression speed

To test the decompression speed, we decompress all inverted lists accessed by the queries from *T09*. In Table 2, we show the average decompression speed for the different integer encoding methods in millions of docIDs per second. Since there is no obvious decoding operation for EF indexes, we omit the decompression speed of EF indexes in Table 2.

Table 2: Decompression speed for different index schemes and document reordering methods

Index Scheme	Speed ($\times 10^6$ docs/sec)			
	URL	FBR	IBDA	TRM
BLK OptPFD	369	365	383	384
BLK VByte	498	491	492	520
GM	361	374	400	402
GM-Run	504	519	550	593

The main observations are the following:

- All index schemes achieve their best improvement on docIDs assigned by TRM, while the lowest speeds are on reordering by FBR. In all reordering methods, GM-Run achieves a better decompression speed than baselines. It yields an enhancement of 1.2% to 14% over the best baseline BLK VByte.
- The decompression speed of GM-Run outperforms GM in all document reordering methods. Unlike for GM-Run, for GM we compress the dictionary with OptPFD, and patterns in runs decompress much faster. The improvement over GM is about 38% to 48%.
- The TRM reordering method can generate longer runs in posting lists, which results in a better decompression speed for GM-Run. More longer patterns also result in fewer random memory accesses caused by pattern fetching during decompression.

From Tables 1 and 2, we can see that an index reordered by FBR can generate better a grammar-based index in terms of compression. For better decompression speed, however, the IBDA and TRM reordering methods perform better. We can conclude that the proposed grammar transformation is able to improve compression ratio by up to 8.8% and decompression speed by up to 14% vs. the baselines.

5.4 Query Processing Performance

We now focus on testing the efficiency of our methods for query processing. Since modern search engines typically utilize multi-core CPUs to boost processing, we not only conduct experiments on a single thread, but also test multi-threaded performance. For multi-threaded experiments, we report both average query processing time and throughput. Higher throughput implies that we can use less hardware resources to support the same number of users or use the same hardware resources to serve more users. For each index scheme, we batch every thousand queries for one new thread to process, while the maximum number of threads is set to 12. The average times here are measured by running the same query set 3 times.

In Table 3(a), we compare the average processing time for AND query processing. We see that grammar indexes underperform both Elias-Fano indexes and block-based indexes with a single thread. The difference between GM and EF ϵ-optimal is about 12% to 26%, while for GM-Run the difference is approximately 1.7% to 10%. Since a grammar index is divided into two parts in the main memory, frequent cache misses on random memory accesses occur during list intersection, which causes the worse performance compared to the baselines. In contrast, the speedups for grammar indexes with multiple threads are better. In this setting, GM underperforms EF ϵ-optimal by 4.3% to 7.5%, and GM-Run even outperforms ϵ-optimal about 3.3% to 15%. The reason is that the cache miss problem for grammar indexes is effectively "hidden" when utilizing multiple threads. When a cache miss occurs in one thread, the CPU instead executes another thread until the cache miss is resolved in the original thread. Therefore, the benefit of GM-Run, i.e., reducing the number of comparisons of docIDs during list intersection (e.g., the difference is about 45.7% over EF ϵ-optimal with index of which docIDs assigned by TRM), results in less query processing time.

The throughput of GM for AND query processing is closer to the baselines, while the throughput for GM-Run is much better than the EF baselines. For GM, the deficiency versus EF ϵ-optimal can be up to 7.5%, although GM slightly outperforms EF ϵ-optimal when docIDs are reassigned by TRM. The improvement of throughput of GM over BLK OptPFD is up to 26%. GM-Run achieves better throughput than most baselines, e.g. it outperforms EF ϵ-optimal by about 14% to 27% and using TRM, it outperforms the best baseline, BLK VByte, by over 5.9%. These observations indicate that the proposed grammar index can achieve higher throughput than the baselines when using multiple threads. Although the average processing time of GM and GM-Run is worse than most baselines, the difference would be insignificant to an end user (typically smaller than 1 ms).

In Tables 3(b) and 3(c), we also compare the processing time and throughput for OR and top-10 WAND queries on the different index schemes. These results indicate the proposed grammar index is not best suited to OR or WAND queries in terms of both query processing time and throughput. Unlike for AND queries, OR and WAND queries do not benefit from identifying patterns, which aims to reduce comparisons during query processing.

Table 3: Average time (ms) and throughput (queries/sec) for different query operations

(a) AND queries

Index Scheme	Avg. time, single thread				Avg. time, multiple threads				Throughput, multiple threads			
	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM
EF Uniform	4.34	3.66	3.48	3.23	0.63	0.55	0.51	0.47	980	1026	1097	1256
EF ϵ-optimal	3.89	3.57	3.11	3.04	0.60	0.53	0.50	0.46	1104	1262	1301	1469
BLK OptPFD	4.11	3.98	3.57	3.42	0.63	0.55	0.52	0.54	1003	1026	1039	1173
BLK VByte	2.82	2.67	2.12	1.99	0.34	0.28	0.24	0.21	1305	1485	1532	1765
GM	4.37	4.04	3.89	3.82	0.64	0.57	0.53	0.48	1021	1214	1300	1483
GM-Run	3.99	3.63	3.43	3.29	0.58	0.50	0.47	0.39	1257	1465	1493	1869

(b) OR queries

Index Scheme	Avg. time, single thread				Avg. time, multiple threads				Throughput, multiple threads			
	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM
EF Uniform	102.2	95.4	91.1	94.3	32.0	31.6	30.8	31.2	31.9	33.9	36.5	35.0
EF ϵ-optimal	104.4	97.8	92.3	96.3	33.4	32.3	30.2	32.2	29.3	32.0	35.6	33.2
BLK OptPFD	101.8	95.4	88.4	98.4	31.4	29.4	23.4	30.9	30.9	33.6	41.8	33.4
BLK VByte	98.4	90.1	86.7	92.3	30.5	28.4	23.2	29.1	34.4	37.8	44.8	37.1
GM	115.9	111.4	106.3	109.8	40.4	38.3	36.9	37.4	25.2	26.9	27.7	27.4
GM-Run	109.7	103.5	97.4	99.9	36.9	35.4	33.2	34.7	27.5	28.6	30.8	29.5

(c) Top-10 WAND queries

Index Scheme	Avg. time, single thread				Avg. time, multiple threads				Throughput, multiple threads			
	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM	URL	FBR	IBDA	TRM
EF Uniform	17.4	14.6	13.4	14.6	4.13	3.87	3.34	3.54	263	286	315	310
EF ϵ-optimal	16.3	12.9	11.4	13.8	3.98	3.58	3.23	3.34	270	289	330	311
BLK OptPFD	14.5	12.4	11.2	13.2	3.72	3.41	3.01	3.14	274	295	321	303
BLK VByte	11.3	9.6	9.2	9.6	3.12	2.98	2.23	2.56	326	346	448	397
GM	21.4	19.3	17.6	16.4	4.32	4.56	4.42	4.33	266	281	271	290
GM-Run	18.4	17.0	14.6	14.0	4.29	4.18	3.91	3.52	259	265	284	309

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new compression scheme where the key idea is generating a context-free grammar to represent the inverted index succinctly and then compressing it using integer encoding. We also present approaches to query processing suited to this grammar-based index scheme.

By evaluating index compression and query processing on various index structures, the integer encoding and document reordering methods yield better general performance on the run-based grammar index structure instead of the traditional inverted index structure. On the grammar index, we obtain up to 8.8% reduction in space usage and up to 14% acceleration in index decompression speed, while still achieving a reasonable query processing performance, especially high throughput with multiple threads condition. As future work, it would be worthwhile using d-gap lists to generate grammar index, and combining grammar based compression method with Elias-Fano coding.

7. ACKNOWLEDGEMENTS

This work is partially supported by NSF of China (61373018, 11301288, 11550110491), Program for New Century Excellent Talents in University (NCET130301) and the Fundamental Research Funds for the Central Universities (65141021).

8. REFERENCES

[1] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. 15th Australasian Database Conference*, volume 27, pages 61–67, Darlinghurst, Australia, 2004.

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

[3] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *Proc. 36th International ACM SIGIR Conference*, pages 173–182, New York, NY, USA, 2013.

[4] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1975.

[5] G. Beskales, M. Fontoura, M. Gurevich, S. Vassilvitskii, and V. Josifovski. Factorization-based lossless compression of inverted indices. In *Proc. of the 20th international*

conference on Information and knowledge management, pages 327–332. ACM, 2011.

[6] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. Data Compression Conference*, pages 342–351, Washington, DC, USA, 2002.

[7] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. 12th International Conference on Information and Knowledge Management*, pages 426–434. ACM, 2003.

[8] S. Büttcher, C. Clarke, and G. V. Cormack. *Information retrieval: Implementing and evaluating search engines*. MIT Press, 2010.

[9] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51:2554–2576, 2005.

[10] J. Chen and T. Cook. Using d-gap patterns for index compression. In *Proc. of the 16th International Conference on World Wide Web*, pages 1209–1210. ACM, 2007.

[11] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. 20th ACM International Conference on Information and Knowledge Management*, pages 463–468, New York, NY, USA, 2011.

[12] F. Claude and J. I. Munro. Document listing on versioned documents. In *Proc. 20th International Symposium on String Processing and Information Retrieval*, pages 72–83, New York, NY, USA, 2013.

[13] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29:1:1–1:25, 2010.

[14] P. Ferragina, F. Piccinno, and R. Venturini. Compressed indexes for string searching in labeled graphs. In *Proc. of the 24th International Conference on World Wide Web*, pages 322–332. ACM, 2015.

[15] G. H. Gonnet, L. D. Rogers, and J. A. George. An algorithmic and complexity analysis of interpolation search. *Acta Informatica*, 13:39–52, 1980.

[16] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. 5th International Conference on Experimental Algorithms*, pages 158–169, Berlin, Heidelberg, 2006.

[17] J. C. Kieffer, E.-H. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46:1227–1245, 2000.

[18] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. In *Proc. Conference on Data Compression*, pages 296–306, Washington, DC, USA, 1999.

[19] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

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

[21] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammar. *The Computer Journal*, 40:103–116, 1997.

[22] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[23] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. of the 8th International Conference on Web Search and Data Mining*, pages 47–56. ACM, 2015.

[24] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM SIGIR Conference*, pages 273–282. ACM, 2014.

[25] M. Petri and A. Moffat. On the cost of phrase-based ranking. In *Proc. of the 38th International ACM SIGIR Conference*, pages 931–934. ACM, 2015.

[26] S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146, 1976.

[27] L. Shi and B. Wang. Yet another sorting-based solution to the reassignment of document identifiers. In *Information Retrieval Technology*, volume 7675 of *Lecture Notes in Computer Science*, pages 238–249, 2012.

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

[29] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. 29th European Conference on IR Research*, pages 101–112, Berlin, Heidelberg, 2007.

[30] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. 27th International ACM SIGIR Conference*, pages 305–312, New York, NY, USA, 2004.

[31] F. Silvestri and R. Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proc. of the 19th international conference on Information and knowledge management*, pages 1219–1228. ACM, 2010.

[32] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. of the 20th international conference on Information and knowledge management*, pages 317–326. ACM, 2011.

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

[34] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, San Francisco, CA, USA, second edition, 1999.

[35] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web*, pages 401–410, New York, NY, USA, 2009.

[36] E.-H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform part one: without context models. *IEEE Transaction on Information Theory*, 46:755–777, 2000.

[37] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. 17th International Conference on World Wide Web*, pages 387–396, New York, NY, USA, 2008.

[38] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. 22nd International Conference on Data Engineering*, pages 59–70, Washington,DC,USA, 2006.