Inverted Index Compression and Query Processing with Optimized Document Ordering

Hao Yan
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
hyan@cis.poly.edu

Shuai Ding
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
sding@cis.poly.edu

Torsten Suel*
Yahoo! Research
701 1st Ave
Sunnyvale, CA 94089
suel@poly.edu

ABSTRACT

Web search engines use highly optimized compression schemes to decrease inverted index size and improve query throughput, and many index compression techniques have been studied in the literature. One approach taken by several recent studies [7, 23, 25, 6, 24] first performs a renumbering of the document IDs in the collection that groups similar documents together, and then applies standard compression techniques. It is known that this can significantly improve index compression compared to a random document ordering.

We study index compression and query processing techniques for such reordered indexes. Previous work has focused on determining the best possible ordering of documents. In contrast, we assume that such an ordering is already given, and focus on how to optimize compression methods and query processing for this case. We perform an extensive study of compression techniques for document IDs and present new optimizations of existing techniques which can achieve significant improvement in both compression and decompression performances. We also propose and evaluate techniques for compressing frequency values for this case. Finally, we study the effect of this approach on query processing performance. Our experiments show very significant improvements in index size and query processing speed on the TREC GOV2 collection of 25.2 million web pages.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval

General Terms

Algorithms, performance

Keywords

Inverted index, search engines, index compression, IR query processing, document ordering

1. INTRODUCTION

Large web search engines need to process thousands of queries per second over tens of billions of pages. Moreover, the results for each query should be returned within at most a few hundred milliseconds. A significant amount of research and engineering has gone into addressing these tremendous performance challenges, and various optimizations have been proposed based on techniques such as caching, data compression, early termination, and massively parallel processing. We focus on one important class of optimizations, index compression. Inverted index compression is used in all major engines, and many techniques have been proposed [26, 29].

Informally, an *inverted index* for a collection of documents is a structure that stores, for each term (word) occurring somewhere in the collection, information about the locations where it occurs. In particular, for each term t, the index contains an *inverted list* I_t consisting of a number of *index postings*. Each posting in I_t contains information about the occurrences of t in one particular document d, usually the ID of the document (the docID), the number of occurrences of t in d (the frequency), and possibly other information about the locations of the occurrences within the document and their contexts. The postings in each list are usually sorted by docID. For example, an inverted list I_t of the form $\{56, 1, 34\}\{198, 2, 14, 23\}$ might indicate that term t occurs once in document 56, at word position 34 from the beginning of the document, and twice in document 198 at positions 14 and 23. We assume postings have docIDs and frequencies but do not consider other data such as positions or contexts.

Many techniques for inverted index compression have been studied in the literature; see [26, 29] for a survey and [1, 2, 3, 30, 27, 14] for very recent work. Most techniques first replace each docID (except the first in a list) by the difference between it and the preceding docID, called *d-gap*, and then encode the d-gap using some integer compression algorithm. Using d-gaps instead of docIDs decreases the average value that needs to be compressed, resulting in a higher compression ratio. Of course, these values have to be summed up again during decompression, but this can usually be done very efficiently. Thus, inverted index compression techniques are concerned with compressing sequences of integers whose average value is small. The resulting compression ratio depends on the exact properties of these sequences, which depend on the way in which docIDs are assigned to documents.

This observation has motivated several authors [7, 23, 25, 6, 24] to study how to assign docIDs in a way that optimizes compression. The basic idea here is that if we assign docIDs such that many similar documents (i.e., documents that share a lot of terms) are close to each other in the docID assignment, then the resulting sequence of d-gaps will become more skewed, with large clusters of many small values interrupted by a few larger values, resulting in better compression. In contrast, if docIDs are assigned at random, the distribution of gaps will be basically exponential, and small values will not be clustered together. In practice, IR systems may assign docIDs to documents in a number of ways, e.g., at random, in the order they are crawled or indexed, or based on global measures of page quality (such as Pagerank [9]).

*Current Affiliation: CSE Dept. Polytechnic Inst. of NYU

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

As we discuss later, in some cases it is difficult or impossible to change the way docIDs are assigned, but there are many other scenarios where reordering of documents could be used to improve index compression.

In this paper, we follow the document reordering approach studied in [7, 23, 25, 6, 24]. However, while previous work has focused on finding the best ordering of documents in a collection, we focus on the next step, how to optimize actual index compression and query processing given some suitable document ordering obtained from previous work. In particular, we extensively study and optimize state-of-the-art compression techniques for docIDs, and propose new algorithms for compressing frequencies, under such optimized orderings. Frequency values tend to be small compared to docID gaps (on average when a word occurs in a web page it occurs only 3 to 4 times), and thus different techniques are needed to improve their compression. We further study the impact of docID reordering on query throughput, and propose and study a new index optimization problem motivated by the trade-off between speed and compression ratio of the various methods. Our experimental results show very significant improvements in both overall index size and query processing speed in realistic settings. To our knowledge, no previous work has looked at frequency compression or query processing performance under the document reordering approach.

The remainder of this paper is organized as follows. In the next section, we provide some technical background and discuss related work. Section 3 describes our contributions in more detail. In Section 4 we study techniques for docID compression, while Section 5 focuses on compression of frequencies. Section 6 evaluates query processing performance, and Section 7 studies hybrid schemes that apply different compression techniques to different lists based on query load. Finally, Section 8 provides concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section, we first outline several known index compression techniques that we use in our work. We then discuss previous work on reordering for better inverted index compression, and discuss the applicability of this approach in real systems. Subsection 2.4 describes block-wise compression and skipping in IR query processors, and Subsection 2.5 introduces the TREC GOV2 data set used by us.

2.1 Index Compression Techniques

Recall that in inverted index compression, our goal is to compress a sequence of integers, either a sequence of d-gaps obtained by taking the difference between each docID and the previous docID, or a sequence of frequency values. In addition, we always deduct 1 from each d-gap and frequency, so that the integers to be compressed are non-negative but do include 0 values. We now provide brief descriptions of some basic index compression techniques to keep the paper self-contained; for more details, please see the cited literature. All methods except IPC were recently implemented and evaluated in [27], and we will reuse and extend these highly tuned implementations.

Var-Byte Coding: Variable-byte compression represents an integer in a variable number of bytes, where each byte consists of one status bit, indicating whether another byte follows the current one, followed by 7 data bits. Thus, $142 = 1 \cdot 2^7 + 16$ is represented as 10000001 0001000, while 2 is represented as 00000010. Var-byte compression does not achieve a very good compression ratio, but is simple and allows for fast decoding [22] and is thus used in many systems.

Rice Coding: This method compresses a sequence of integers by first choosing a b such that 2^b is close to the average value. Each integer n is then encoded in two parts: a quotient $q = \lfloor n/(2^b) \rfloor$ stored in unary code using $q + 1$ bits, and a remainder $r = n \bmod 2^b$ stored in binary using b bits. Rice coding achieves very good compression on standard unordered collections but is slower than var-byte, though the gap in speed can be reduced by using an optimized implementation described in [27].

S9: Simple9 coding is an algorithm proposed in [2] that combines good compression and high decompression speed. The basic idea is to try to pack as many values as possible into a 32-bit word. This is done by dividing each word into 4 status bits and 28 data bits, where the data bits can be partitioned in 9 different ways. For example, if the next 7 values are all less than 16, then we can store them as 7 4-bit values. Or if the next 3 values are less than 512, we can store them as 3 9-bit values (leaving one data bit unused).

Simple9 uses 9 ways to divide up the 28 data bits: 28 1-bit numbers, 14 2-bit numbers, 9 3-bit numbers (one bit unused), 7 4-bit numbers, 5 5-numbers (three bits unused), 4 7-bit numbers, 3 9-bit numbers (one bit unused), 2 14-bit numbers, or 1 28-bit numbers. The 4 status bits store which of the 9 cases is used. Decompression can be optimized by hardcoding each of the 9 cases using fixed bit masks, and using a switch operation on the status bits to select the case.

S16: Simple16 (S16) [27] uses the same basic idea as S9, but has 16 ways of partitioning the data bits, where each of the 16 cases uses all 28 data bits. The result is that S16 approximately matches the speed of S9, while achieving slightly better compression. We note here that there are other methods related to S9, such as *Relate10* and *Carryover12* [2], that also achieve improvements over S9 in certain cases.

PForDelta: This is a compression method recently proposed in [14, 30] that supports extremely fast decompression while also achieving a small compressed size. PForDelta (PFD) first determines a value b such that most of the values to be encoded (say, 90%) are less than 2^b and thus fit into a fixed bit field of b bits each. The remaining values, called *exceptions*, are coded separately. If we apply PFD to blocks containing some multiple of 32 values, then decompression involves extracting groups of 32 b-bit values, and finally patching the result by decoding a smaller number of exceptions. This process can be implemented extremely efficiently by providing, for each value of b, an optimized method for extracting 32 b-bit values from b memory words. PFD can be modified and tuned in various ways by choosing different thresholds for the number of exceptions allowed, and by encoding the exceptions in different ways. We use some modifications to PFD proposed in [27], but also add in this paper additional ones that achieves significantly better performance in terms of both size and speed.

Interpolative Coding: This is a coding technique proposed in [17] that is ideal for the types of clustered or bursty term occurrences that exist in real large texts (such as books). In fact, the goal of the document reordering approach is to create more clustered, and thus more compressible, term occurrences, and Interpolative Coding (IPC) has been shown to perform well in this case [6, 7, 23, 24, 25].

IPC differs from the other methods in an important way: It directly compresses docIDs, and not docID gaps. Given a set

of docIDs $d_i < d_{i+1} < \ldots < d_j$ where $l < d_i$ and $d_j < r$ for some bounding values l and r known to the decoder, we first encode d_m where $m = (i + j)/2$, then recursively compress the docIDs $d_i, \ldots, d_{m-1}$ using l and d_m as bounding values, and then recursively compress $d_{m+1}, \ldots, d_j$ using d_m and r as bounding values. Thus, we compress the docID in the center, and then recursively the left and right half of the sequence. To encode d_m, observe that $d_m > l + m - i$ (since there are $m - i$ values $d_i, \ldots d_{m-1}$ between it and l) and $d_m < r - (j - m)$ (since there are $j - m$ values $d_{m+1}, \ldots d_j$ between it and r). Thus, it suffices to encode an integer in the range $[0, x]$ where $x = r - l - j + i - 2$ that is then added to $l + m - i + 1$ during decoding; this can be done trivially in $\lceil \log_2(x + 1) \rceil$ bits, since the decoder knows the value of x.

In areas of an inverted list where there are many documents that contain the term, the value x will be much smaller than $r - l$. As a special case, if we have k docIDs larger than l and less than r where $k = r - l - 1$, then nothing needs to be stored at all as we know that all docIDs properly between l and r contain the term. This also means that IPC can use less than one bit per value for dense term occurrences.

Evaluation: Index compression techniques are usually evaluated in terms of: (1) The compression ratio, which determines the amount of main memory needed for a memory-based index or the amount of disk traffic for a disk-based index. State-of-the-art systems typically achieve compression ratios of about 3 to 10 versus the naive 32-bit representation, while allowing extremely fast decompression during inverted list traversals. (2) The decompression speed, typically hundreds of millions of integers per second, which is crucial for query throughput. In contrast, compression speed is somewhat less critical, since each inverted list is compressed only once during index building, and then decompressed many times during query processing.

We note that there are two different ways to evaluate the compression ratio. We can consider the total size of the index; this models the amount of space needed on disk, and also the amount of main memory needed if the index is held entirely in main memory during query processing. Alternatively, we can measure the compressed size of the inverted lists associated with an average query under some query load; this models the amount of data that has to be transferred from disk for each query if the index is entirely on disk (and also the amount of data that has to be moved from main memory to CPU as this can become a bottleneck in highly optimized systems). In reality, most systems cache part of the index in memory, making a proper evaluation more complicated. We consider both cases in our experiments, but find that the relative ordering of the algorithms stays the same.

2.2 Document Reordering and Related Ideas

Several papers have studied how to reorder documents for better compression [7, 23, 25, 24, 6]. In particular, the approaches in [7, 23, 25, 6] first perform some form of text clustering on the collection to find similar documents, and then assign docIDs by traversing the resulting graph of document similarities in a Depth-First-Search [7] or TSP-like fashion. Subsequent work in [24] looked at a much simpler approach, assigning docIDs alphabetically according to URL, and showed that this method basically matches the performance of previous techniques based on text clustering. Note that such an alphabetical ordering places all documents from the same site, and same subdirectory within a site, next to each other. This results in improved compression as such documents often have the same topic and writing style.

We use alphabetical assignment of docIDs in all experiments, but our techniques work with any ordering. Our focus is not on finding a better assignment, but on exploiting an existing assignment using optimized compression and query processing techniques. In contrast, previous work considered only a few standard techniques for docID compression, and did not consider frequency compression or query processing.

Another related problem is the compression of inverted indexes for archival collections, i.e., collections that contain different versions of documents over a period of time, with often only minor changes between versions. This problem has recently received some attention in the research community [11, 15, 28, 5], and the basic idea is also to exploit similarity between documents (or their versions). The techniques used are different, and more geared towards getting very large benefits for collections with multiple very similar versions, as opposed to the reordering approach here which tries to exploit more moderate levels of similarity. In future work, it would be very interesting to compare these different approaches on documents with different degrees of similarity. For example, the alphabetical ordering used here could be easily extended to versioned collections (by sorting first by URL and then by version number), and could in fact be seen as providing an alternative efficient implementation of the approach in [5] that is based on merging consecutive postings in a list.

2.3 Feasibility of Document Reordering

IR systems may assign docIDs to documents in a number of ways, e.g., at random, in the order they are crawled or indexed, or sometimes based on global measures of page quality (such as Pagerank [9]) that can enable faster query processing through early termination. The document reordering approach in this paper and the previous work in [7, 23, 25, 6, 24] assumes that we can modify this assignment of docIDs to optimize compression. While this is a reasonable assumption for some systems, there are other cases where this is difficult or infeasible. We now discuss two cases, distributed index structures, and tiering and early termination techniques.

Large-scale search engines typically partition their document collection over hundreds of nodes and then build a separate index on each node. If the assignment of documents to nodes is done at random, then a local reordering of documents within a node might not give much benefit. On the other hand, if pages are assigned to nodes based on a host-level assignment or alphabetical range-partitioning, then we would expect significant benefits. However, this might require changes in the architecture and could impact issues such as load balancing.

Document ordering is also complicated by the presence of tiering and other early termination mechanisms, which are widely used in current engines. In a nutshell, these are techniques that avoid a full traversal of the inverted lists for most queries through careful index layout, which often involves some reordering of the documents. In some approaches, such as a document-based tiering approach [21], or a partitioning of inverted lists into a small number of chunks [19, 16], reordering for better compression can be applied within each tier or chunk. Other approaches may assign docIDs based on Pagerank [9] or other global document scores mined from the collection [20], or use a different ordering for each list [13]; in these cases our approach may not apply.

2.4 Query Processing in Search Engines

Query processing in state-of-the-art systems involves a number of phases such as query parsing, query rewriting, and the computation of complex, often machine-learned, ranking functions that may use hundred of features. However, at the lower layer, all such systems rely on extremely fast access to an inverted index to achieve the required query throughput. In particular, for each query the engine typically needs to traverse the inverted lists corresponding to the query terms in order to identify a limited set of promising documents that can then be more fully scored in a subsequent phase. The challenge in this initial filtering phase is that for large collections, the inverted lists for many commonly queried terms can get very long. For example, for the TREC GOV2 collection of 25.2 million web pages used in this paper, on average each query involves lists with several million postings.

Current systems typically use a style of query processing called *document-at-a-time* (DAAT) query processing, where all inverted lists associated with a query are opened for reading and then traversed in an interleaved fashion. This approach has several advantages: (a) it performs extremely well on the AND and WAND [10] style queries common in search engines, (b) it enables a very simple and efficient interface between query processing and the lower-level index decompression mechanism, and (c) it allows for additional performance gains through forward skips in the inverted lists, assuming that the postings in each list are organized into blocks of some small size that can be independently decompressed.

In our experiments, we use an optimized DAAT query processor developed in our group, and we organize each inverted list into blocks with a fixed number of postings. We choose 128 postings as our default block size (shown to perform well, e.g., in [27]), and keep for each inverted list two separate arrays containing the last docID and size of each block in words in (almost) uncompressed form. This allows skipping of blocks during query processing by searching in the array of last docIDs. All decompression is performed in terms of blocks; to add another compression method to our query processor it suffices to supply a method for uncompressing the docIDs of a block, and one to uncompress the frequencies. (A block consists of all 128 docIDs followed by all 128 frequency values.) This design is highly useful in Section 7, where we use several compression techniques within the same index.

One interesting result of our experiments is that reordering of documents, in addition to improving compression, also speeds up index traversal in a DAAT query processor. In particular, our query processor (with no changes in the software, and independent of compression method) performs more and larger forward skips during index access in the reordered case, and as a result decompresses less than half as many blocks per query as in the unordered case. Note that this is related to, but different from, recent work in [8, 12] that shows how to choose an optimal set of forward pointers (basically, how to choose variable block boundaries) for each list based on an analysis of the query load. Thus, we reorder documents while keeping block sizes constant, while [8, 12] modify block sizes while keeping the ordering constant; it would be interesting to see how the approaches work in combination, and whether the reordering could be improved by considering query loads.

2.5 The TREC GOV2 Data Set

For our experiments, we use the TREC GOV2 data set of 25.2 million web pages from the `gov` domain that is distributed by the US National Institute of Standards and Technology (NIST) and used in the annual TREC competitions. This data is widely used for research in the IR community, thus allowing others to replicate our results. It is based on a 2004 crawl of the `gov` domain, and is also accompanied by a set of 100000 queries (the 2006 Efficiency Task Topics) that we use in our evaluation.

While the data set does not represent a complete snapshot of the `gov` domain at the time of the crawl, it nonetheless contains a fairly significant subset of it. This is important since our techniques perform best on "dense" data sets such as GOV2 that are based on a fairly deep crawl of a subset of domains. In contrast, a "sparse" set of 25.2 million pages crawled at random from the many billions of pages on the web would not benefit as much.

3. CONTRIBUTIONS OF THIS PAPER

In this paper, we study the problem of optimizing compression and query processing performance given a suitable assignment of docIDs. Previous work in [7, 23, 25, 6, 24] focused on finding a good docID assignment, and then evaluated the assignment by compressing docIDs using standard techniques. In contrast, we focus on how to best exploit a given assignment by optimizing compression and query processing techniques for this case. Our compression codes are available at `http://cis.poly.edu/westlab/`. Our main contributions are as follows:

(1) We propose new versions of the PForDelta (PFD) approach and compare them with state-of-the-art techniques in the literature as well as new variants that are tuned for both speed and compression ratio. Our experimental results show that our versions of PFD can achieve significant improvements in size and speed.

(2) We study the compression of frequency values under such assignments. Previous work only considered docIDs, but we show that frequencies can also be compressed significantly better through suitable docID assignment. Our main contribution here is the application of transformations inspired by move-to-front coding to improve the compressibility of frequency values.

(3) We study the impact of docID reordering on overall index size and query throughput on the TREC GOV2 data set of 25.2 million web pages. We observe a reduction in minimum index size by about 50% over the case of a random docID ordering, resulting in a minimal size of about 3.45 GB for a full-text index of the entire collection. We also show that the docID reordering leads to significant improvements in query throughput on conjunctive queries for document-at-a-time (DAAT) query processors by reducing the number of random seeks in the index, in addition to any benefits obtained via the reduction in index size.

(4) The various compression techniques studied by us show a trade-off between speed and compression ratio. Thus, the techniques that achieve the smallest size are much slower than the fastest ones, which in turn result in a larger index size. This motivates us to study hybrid index organizations that apply different compression schemes to different lists. We set up a formal optimization problem and show that by selecting a suitable compression scheme for each list based on an analysis of a query log, we can simultaneously achieve almost optimal size and speed.

4. DOCID COMPRESSION

In this section, we perform a detailed study of compression techniques for docIDs. In particular, we first study distributions of docIDs on TREC GOV2 data set, and then discuss state-of-the-art compression methods and propose our new algorithms, and finally we evaluate all these methods through some preliminary experiments.

4.1 Distributions of DocIDs

The performance of a compression method depends on the data distribution it is applied to. For inverted index compression, compression is best when there are many small numbers. The optimized assignment of docIDs is intended to increase the number of small numbers and thus improve compression performance. In Figure 1, we show a histograms of d-gaps for the TREC GOV2 data set under three different orderings of documents: *original*, which we get from the official TREC GOV2 data set; *sorted*, where docIDs are re-assigned by us after we sort their URLs, as in [24]; and *random*, where docIDs are assigned at random.

From Figure 1 we can see that the *sorted* ordering results in more small gaps than the other two kinds of indexes, suggesting a higher compression ratio. In addition, the d-gaps for the *original* ordering have a similar histogram as those for the *random* ordering, suggesting that the compression methods will very likely have a similar performance. Furthermore, we analyze individual inverted lists and find that such a reordering results in more clusters (not shown in the Figure 1), i.e., sequences of consecutive small d-gaps.

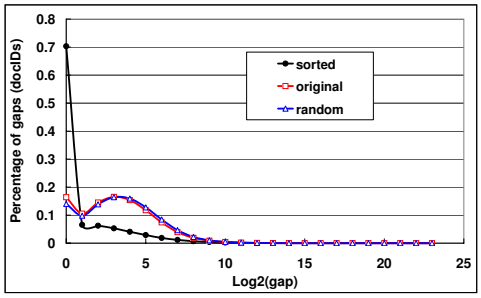

Figure 1: Histograms of d-gaps for inverted lists corresponding to 1000 random queries on the TREV GOV2 data set, under three different orderings: *original*, *sorted* and *random*. The x-axis is the number of bits required to represent d-gaps in binary, and the y-axis is the percentage of such d-gaps. (Thus, the first point is for 1-gaps, the second for 2-gaps, the third for 3-gaps plus 4-gaps.)

4.2 Optimizing PForDelta compression

We now describe two modifications to PFD that achieve significant improvements over the versions in [30, 14, 27]. Recall that the implementations of PFD in previous work encode a block of 128 value by first allocating 128 b-bit slots, and then for those 90% of the values less than 2^b directly storing them in their corresponding slots. For each value larger than 2^b, called a *exception*, we store an offset value in the exception's corresponding slot indicating the distance from the current exception to the next one, and the actual value of the exception in some additional space after the 128 b-bit slots. One disadvantage of such a code structure is that when two consecutive exceptions have a distance of more than 2^b, we have to use more than one offset to represent the distance, by forcing additional exceptions in between these two exceptions. We cannot solve this problem by simply increasing b since this would waste lots of bits on 90% of values; but if we decrease b more exceptions will be produced. This means in particular that this version of PFD cannot profitably use any values of b less than $b = 3$, but this case is very important in the reordered case.

To overcome this problem, we present a new code structure for PFD that stores the offset values and parts of the exceptions in two additional arrays. In particular, for an exception, we store its lower b bits, instead of the offset to the next exception, in its corresponding b-bit slot, while we store the higher *overflow* bits and the offset in two separate arrays. These two arrays can be further compressed by any compression method, and we find that S16 is particularly suitable for this. We call this approach NewPFD.

Our second improvement is in the selection of the b value for each block. As it turns out, selecting a constant threshold for the number of exceptions does not give the best tradeoff between size and speed. Instead, we model the selection of the b for each block as an optimization problem similar to that in Section 7. Thus, we initially assign the b with the smallest compressed size to each block, and then increase speed as desired by selecting a block that gives us the most time savings per increase in size, and change the b of that block. We call this OptPFD. We note here that for a given target speed, we can easily derive simple global rules about the choice of b, instead of running the iterative optimization above. Thus this version can be very efficiently implemented even on very large collections.

4.3 Optimizing other methods

We now present a few minor optimizations of some other methods that we used in our experimental evaluation.

GammaDiff: This is a variation of Gamma coding that stores an integer x by encoding the unary part of the Gamma code (that is, $1+\lfloor logx \rfloor$) as the difference between $1+\lfloor logx \rfloor$ and the number of bits required to represent the average of all gaps in the list. The motivation is that when docIDs are clustered, the differences between d-gaps and their average gap may be smaller than the gaps.

S16-128: As S9 and S16 only have 9 or 16 possible cases for encoding numbers, sometimes they have to choose a wasteful case when a better one might exist. Now suppose we have a sequence of numbers consisting mainly of small values. In this case, a version of S16 called S16-128 can do slightly better by providing more cases for small numbers and fewer for larger numbers.

Optimized IPC: Recall that the key step of interpolative coding (IPC) is to encode a number x in the range $< lo, hi >$, where lo and hi are respectively the lowest and highest possible values of x. The original IPC encodes the offset $o = x - lo$ using a b-bit number, where $b = \lceil r \rceil$ and $r = hi - lo + 1$ is the number of possible values of the offset. This wastes bits if r is not a power of 2. We can do better by using a trick from Golomb coding to encode o as follows: If $o < 2^b - r$, use $b-1$ bits to represent o, otherwise use b bits to represent $o + 2^b - r$. (This technique was already described for IPC in [26].) In addition, before we apply the above optimization, we transform the range of values in such a way that the shorter codes are applied to values in the middle of the range, since such values are more likely even in a highly clustered list. Also, while IPC is usually considered as a list-oriented method,

meaning it starts by encoding the median of the entire list, we apply it to blocks of a certain size. As it turns out, this also improves compression if we choose a good block size. In particular, block sizes of the form $2^b - 1$ appear to work best, and thus we usually choose blocks of size 127.

4.4 Preliminary Experiments

We first describe our experimental setup, which we also use in later sections. For the data set, we used the TREC GOV2 data set. We then selected 1000 random queries from the supplied query logs; these queries contain 2171 unique terms. All experiments were performed on a single core of a 2.66GHz Intel(R) Core(TM)2 Duo CPU with 8GB of memory.

	sorted	original	random
list-IPC w/o opt	**0.95**	2.70	2.83
list-IPC	**0.88**	2.46	2.57
block-IPC	**0.85**	2.40	2.51

Table 1: Compressed size in MB/query for docIDs using a basic list-wise IPC (no optimizations), a list-wise version with optimizations enabled, and its block-wise version, under the original, sorted, and random orderings.

In Table 1, we compare the original IPC, which is list-wise, with its improved version with our various optimizations and its block-wise version with our optimizations, on the GOV2 data set under the original, sorted, and random orderings. From Table 1, we can observe the following: First, all IPC algorithms work significantly better on the d-gaps under the *sorted* ordering than under the other two orderings; second, both list-wise and block-wise IPC with our optimizations are much better the original IPC, but block-wise IPC with our optimizations achieves the best compression.

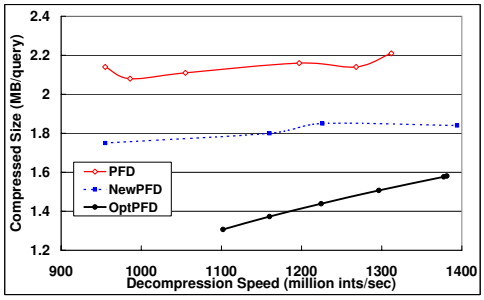

Figure 2: Compressed size in MB/query versus decompression speed in million integers per second for docIDs, using PFD, NewPFD, and OptPFD under the sorted ordering. The points from left to right for PFD and NewPFD correspond to the following percentages of exceptions: 5%, 8%, 10%, 20%, and 30%. For OptPFD, the points correspond to different target speeds for the optimization and their corresponding sizes.

Compared to IPC, the main advantage of PFD is that decoding is very fast. In Figure 2, we show the trade-offs between decompression speed and compressed size for PFD, NewPFD, and OptPFD as introduced above. From Figure 2, we see that OptPFD can always achieve a much smaller compressed size for a given decoding speed than the other method. Thus, choosing b not based on a global threshold on exceptions, but based on a global target speed, achieves a much better trade-off than the naive global threshold used in PFD and NewPFD. While OptPFD is still worse than IPC in terms of compressed size, decompression is much faster than for any version of IPC (as we will show later). We also ran experiments under the original document ordering, and observed slightly smaller but still significant gains for OptPFD over PFD and NewPFD, while PFD and newPFD were overall similar in performance.

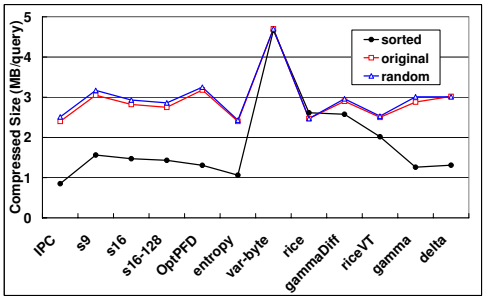

Figure 3: Compressed size in MB/query for docIDs under the original, sorted, and random orderings.

In Figure 3, we compare the average compressed size per query of the docIDs for most of the state-of-the-art inverted index compression methods on the TREC GOV2 data set under the original, sorted, and random orderings. For each data set, we show results of twelve compression methods: var-byte, S9, S16, S16-128, OptPFD, Delta coding, Gamma coding, GammaDiff, Rice coding, a variant of Rice coding called RiceVT described in [26, 18] which essentially promotes the implicit probabilities of small gaps, the block-wise interpolative coding with our above optimizations, and entropy, which uses the global frequency distribution of the compressed integers. For OptPFD, we chose a setting that minimizes the compressed size.

From Figure 3, we make the following observations: First, just as Figure 1 suggested, many compression methods can achieve a much better compression ratio on the d-gaps under the *sorted* ordering than under the other two orderings; second, all compression methods on d-gaps under the *original* ordering achieve similar performances with those under the *random* orderings; third, IPC achieves the best compression performance among all methods; fourth, OptPFD is quite competitive with all other methods (even with IPC, although it is slightly worse than IPC in terms of size). One disadvantage of IPC is that its decompression is slow. In contrast, all other methods to the left of the *entropy* method are fairly fast, and much faster than those further to the right.

5. FREQUENCY COMPRESSION

In this section, we first discuss the effect of docID reordering on frequencies, and then propose more effective compression algorithms. In particular, we show that reordered frequencies can be transformed in such a way that their entropy is lowered significantly, leading to better compression.

5.1 Effect of Reordering on Frequencies

Frequency values by themselves are not changed at all by reordering, and thus reassigning docID by sorting URLs does not affect the distribution of frequencies. However, such an ordering results in more local clusters of similar values. This can be shown by comparing the compressed size of context-sensitive and context-free methods. The former methods, which include IPC, S9, S16, and OptPFD, encode batches of numbers, while the latter methods, such as gamma or delta coding, encode each number independently, resulting in no change in compression after reordering.