Performance of Compressed Inverted List Caching in Search Engines*

Jiangong Zhang
CIS Department
Polytechnic University
Brooklyn, NY 11201, USA
zjg@cis.poly.edu

Xiaohui Long
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
xiaohui.long@microsoft.com

Torsten Suel
CIS Department
Polytechnic University
Brooklyn, NY 11201, USA
suel@poly.edu

ABSTRACT

Due to the rapid growth in the size of the web, web search engines are facing enormous performance challenges. The larger engines in particular have to be able to process tens of thousands of queries per second on tens of billions of documents, making query throughput a critical issue. To satisfy this heavy workload, search engines use a variety of performance optimizations including index compression, caching, and early termination.

We focus on two techniques, inverted index compression and index caching, which play a crucial rule in web search engines as well as other high-performance information retrieval systems. We perform a comparison and evaluation of several inverted list compression algorithms, including new variants of existing algorithms that have not been studied before. We then evaluate different inverted list caching policies on large query traces, and finally study the possible performance benefits of combining compression and caching. The overall goal of this paper is to provide an updated discussion and evaluation of these two techniques, and to show how to select the best set of approaches and settings depending on parameter such as disk speed and main memory cache size.

Categories and Subject Descriptors

H.3.1 [**Information Systems**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Systems**]: Information Search and Retrieval—*Search process.*

General Terms

Performance, Experimentation.

Keywords

Search engines, inverted index, index compression, index caching.

1. INTRODUCTION

Web search engines are probably the most popular tools for locating information on the world-wide web. However, due to the rapid growth of the web and the number of users, search engines are faced with formidable performance challenges. On one hand, search engines have to integrate more and more advanced techniques for tasks such as high-quality ranking, personalization, and spam detection. On the other hand, they have to be able to process tens of thousands of queries per second on tens of billions of pages; thus query throughput is a very critical issue.

In this paper, we focus on the query throughput challenge. To guarantee throughput and fast response times, current large search engines are based on large clusters of hundreds or thousands of servers, where each server is responsible for searching a subset of the web pages, say a few million to hundreds of millions of pages. This architecture successfully distributes the workload over many servers. Thus, to maximize overall throughput, we need to maximize throughput on a single node, still a formidable challenge given the data size per node. Current search engines use several techniques such as index compression, index caching, result caching, and query pruning (early termination) to address this issue.

We consider two important techniques that have been previously studied in web search engines and other IR systems, inverted index compression and inverted index caching. Our goal is to provide an evaluation of state-of-the-art implementations of these techniques, and to study how to combine these techniques for best overall performance on current hardware. To do this, we created highly optimized implementations of existing fast index compression algorithms, including several new variations of such algorithms, and evaluated these on large web page collections and real search engine traces. We also implemented and evaluated various caching schemes for inverted index data, and studied the performance gains of combining compression and caching depending on disk transfer rate, cache size, and processor speed. We believe that this provides an interesting and up-to-date picture of these techniques that can inform both system developers and researchers interested in query processing performance issues.

2. TECHNICAL BACKGROUND

Web search engines as well as many other IR systems are based on an inverted index, which is a simple and efficient data structure that allows us to find all documents that contain a particular term. Given a collection of N documents, we assume that each document is identified by a unique *document ID* (docID) between 0 and $N-1$. An inverted index consists of many inverted lists, where each inverted list I_w is a list of postings describing all places where term w occurs in the collection. More precisely, each posting contains the docID of a document that contains the term w, the number of occurrences of w in the document (called *frequency*), and sometimes also the exact locations of these occurrences in the document (called *positions*), plus maybe other context such as the font size of the term etc. The postings in an inverted list are typically sorted by docID, or sometimes some other measure.

We consider two cases: (i) the case where we have docIDs and frequencies, i.e., each posting is of the form (d_i, f_i), and (ii) the case where we also store positions, i.e., each posting is of the form $(d_i, f_i, p_{i,0}, \ldots, p_{i,freq-1})$. We use word-oriented positions, i.e., $p_{i,j} = k$ if a word is the k-th word in the document. For many well-known ranking functions, it suffices to store only docIDs and frequencies, while in other cases positions are needed.

On first approximation, search engines process a search query "dog, cat" by fetching and traversing the inverted lists for "dog" and "cat". During this traversal, they intersect (or merge) the postings from the two lists in order to find documents containing all (or at least one) of the query terms. At the same time, the engine also

*Research supported by NSF ITR Award CNS-0325777. Work by the second author was performed while he was a PhD student at Polytechnic University.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

computes a ranking function on the qualifying documents to determine the top-k results that should be returned to the user. This ranking function should be efficiently computable from the information in the inverted lists (i.e., the frequencies and maybe positions) plus a limited amount of other statistics stored outside the inverted index (e.g., document lengths or global scores such as Pagerank). In this paper, we are not concerned with details of the particular ranking function that is used. Many classes of functions have been studied; see, e.g., [5] for an introduction and overview.

2.1 Compressed Index Organization

In large search engines, each machine typically searches a subset of the collection, consisting of up to hundreds of millions of web pages. Due to the data size, search engines typically have to store the inverted index structure on disk. The inverted lists of common query terms may consist of millions of postings. To allow faster fetching of the lists from disk, search engines use sophisticated compression techniques that significantly reduce the size of each inverted list; see [26] for an overview, and [1, 11, 27, 21] for recent methods that we consider in this paper. We describe some of these techniques in more detail later. However, the main idea is that we can compress docIDs by storing not the raw docID but the difference between the docIDs in the current and the preceding posting (which is a much smaller number, particularly for very long lists with many postings). We can compress the frequencies because most of these values are fairly small as a term often occurs only once or twice in a particular document. In general, inverted list compression benefits from the fact that the numbers that need to be coded are on average small (though some may be larger).

We now describe in more detail how to organize a compressed inverted index structure on disk, using as example our own high-performance query processor. We believe that this is a fairly typical organization on disk, though the details are not often described in the literature. The overall structure of our index is shown in Figure 1. The index is partitioned into a large number of *blocks*, say of size 64 KB. An inverted list in the index will often stretch across multiple blocks, starting somewhere in one block and ending somewhere in another block. Blocks are the basic unit for fetching index data from disk, and for caching index data in main memory.

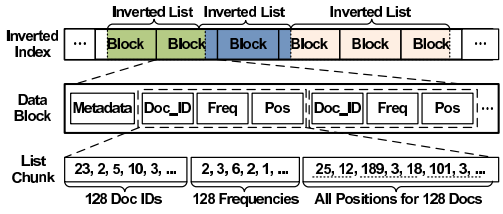

Figure 1: Disk-based inverted index structure with blocks for caching and chunks for skipping. DocIDs and positions are shown after taking the differences to the preceding values.

Thus, each block contains a large number of postings from one or more inverted lists. These postings are again divided into *chunks*. For example, we may divide the postings of an inverted list into chunks with 128 postings each. A block then consists of some meta data at the beginning, with information about how many inverted lists are in this block and where they start, followed by a few hundred or thousand such chunks. In each chunk of 128 postings, it is often beneficial to separately store the 128 docIDs, then the 128 corresponding frequency values, followed by the position values if applicable. Chunks are our basic unit for decompressing inverted index data, and decompression code is tuned to decompress a chunk in fractions of a microsecond. (In fact, this index organization allows us to first decode all docIDs of a chunk, and then later the frequencies or positions if needed.)

During query processing, it is often beneficial to be able to seek forward in an inverted list without reading and decoding all postings, and inverted index structures often store extra pointers that allow us to skip over many of the postings; see, e.g., [18, 19, 7, 8]. This is easily achieved as follows: For each chunk, we separately store the size of this chunk (in bytes or words), and in uncompressed form the largest (last) docID in the chunk. This data can be stored in separate smaller arrays in main memory, or in the meta data at the beginning of each block. This allows skipping over one or more chunks of postings by comparing the docID we are searching for to the uncompressed values of the largest docIDs in the chunks; if the desired value is larger, we can skip the chunk. We note that instead of having a constant number of postings per chunk, one could also fix the size of each chunk (say to 256 bytes or one CPU cache line) and then try to store as many postings as possible in this space. This has the advantage that we can align chunks with cache line boundaries, but for many compression schemes there are some resulting complications, especially if positions are also stored.

In our implementation and experiments, we use this index organization on disk. For simplicity, we assume that during query processing, all chunks of the relevant inverted lists have to be decoded. In reality, we may only have to decode half or less of all the chunks, as we may be able to skip over the other chunks. However, the possible benefit due to skipping depends on the ranking function, which is largely orthogonal to our work. We believe that our assumption provides a good approximation of the real decompression cost, which could be easily adjusted depending on the percentage of chunks that are actually decoded[1].

2.2 Caching in Search Engines

Caching is a standard technique in computer systems, and search engines use several levels of caching to improve query throughput. The most common mechanisms are result caching and list caching [20]. Result caching basically means that if a query is identical to another query (by possibly a different user) that was recently processed, then we can simply return the previous result (assuming no major changes in the document collection and no complications due to localization or personalization). Thus, a result cache stores the top-k results of all queries that were recently computed by the engine. List caching (or inverted index caching), on the other hand, caches in main memory those parts of the inverted index that are frequently accessed. In our case, list caching is based on the blocks described in the previous subsection. If a particular term occurs frequently in queries, then the blocks containing its inverted list are most likely already in cache and do not have to be fetched from disk. Search engines typically dedicate a significant amount of their main memory to list caching, as it is crucial for performance. Note that the cached data is kept in compressed form in memory since the uncompressed data would typically be about 3 to 8 times larger depending on index format and compression method; as we will see decompression can be made fast enough to justify this decision.

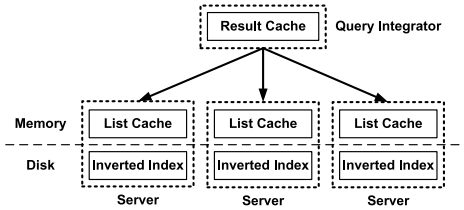

Figure 2: Two-level architecture with result caching at the query integrator and inverted index caching in the memory of each server.

[1]For example, for conjunctive queries our query processor decoded the docIDs of slightly more than half, and the frequencies of only about a third, of all chunks. But for other types of queries the numbers would be higher, assuming no early termination.

Figure 2 shows a simple parallel search engine architecture with two levels of caching. Queries enter the engine via a *query integrator* that first checks its local result cache. If the result is in the cache, it is returned to the user. Otherwise, the query integrator broadcasts the query to a number of machines, each responsible for a subset of the collection. Each machine computes and return the top-k results on its own data, and the query integrator determines the global top-k results. In this architecture, list caching takes places at each machine, which runs its own caching mechanism independent of the other machines. In this paper, we focus on the list caching mechanism. However, we account for the existence of result caching by removing from our traces any duplicate queries that would usually be handled by result caching. Note that keeping these duplicate queries would result in better but unrealistic hit rates.

The performance benefits of result caching were previously studied in [15, 23, 13, 6, 10, 3, 4]. The benefits can vary significantly between search engines, however, based on whether term ordering in queries is considered or stemming is performed. Early work on index caching in IR systems appears in [12], but with a somewhat different setup from that of current search engines. Two-level caching, i.e., the combination of result caching and list caching, was studied in [20], where an LRU policy was used for list caching, and in [6, 3], where several methods are compared but under somewhat different architectural models and objective functions. We will discuss these differences in Section 5, where we compare LRU with several other caching policies and show that there are in fact much better list caching policies than LRU for typical search engine query traces under our model. Finally, recent work in [14] proposes a three-level caching scheme that inserts an additional level of caching between result caching and list caching; we do not consider this scheme here.

3. CONTRIBUTIONS OF THIS PAPER

We study the problem of improving the query throughput in large search engines through use of index compression and index caching techniques. We implement a number of different techniques, including some new variations, and show that very significant performance benefits can be obtained. In particular, our contributions are:

(1) We perform a detailed experimental evaluation of fast state-of-the-art index compression techniques, including Variable-Byte coding [21], Simple9 coding [1], Rice coding [26], and PForDelta coding [11, 27]. Our study is based on highly tuned implementations of these techniques that take into account properties of the current generation of CPUs.

(2) We also describe several new variations of these techniques, including an extension of Simple9 called Simple16, which we discovered during implementation and experimental evaluation. While algorithmically not very novel, these variations give significant additional performance benefits and are thus likely to be useful.

(3) We compare the performance of several caching policies for inverted list caching on real search engine query traces (AOL and Excite). Our main conclusion is that LRU is not a good policy in this case, and that other policies achieve significantly higher cache hit ratios.

(4) Our main novel contribution is a study of the benefits of combining index compression and index caching which shows the impact of compression method, caching policy, cache size, and disk and CPU speeds on the resulting performance. Our conclusion is that for almost the entire range of system parameters, PForDelta compression with LFU caching achieves the best performance, except for small cache sizes and fairly slow disks when our optimized Rice code is slightly better.

4. INVERTED INDEX COMPRESSION

In this section, we study index compression methods and their performance. We first describe the methods we consider and our experimental setup, and then evaluate the performance of the various methods. We note that there are a large number of inverted index compression techniques in the literature; see, e.g., [26] for an overview. We limit ourselves to techniques that allow very fast decompression, say, in excess of a hundred million integers per second. While techniques such as Gamma or Delta coding or various local models are known to achieve good compression [26], they are unlikely to be used in web search engines and other large scale IR systems due to their much slower decompression speeds.

4.1 Algorithms for List Compression

We implemented five different compression methods, Variable-Byte coding (var-byte) [21], Simple9 (S9) [1], a new extension called Simple16 (S16), PForDelta [11, 27], and the classical Rice coding [26]. We now describe each of these algorithms briefly, and then discuss our implementation. In all the methods, we assume that the docID, frequency, and position values in each posting $p_i = (d_i, f_i, p_{i,0}, \ldots, p_{i,f_i-1})$ have been preprocessed as follows before coding: Each docID d_i with $i > 0$ is replaced by $d_i - d_{i-1} - 1$, each f_i is replaced by $f_i - 1$ (since no posting can have a frequency of 0), and each $p_{i,j}$ with $j > 0$ is replaced by $p_{i,j} - p_{i,j-1} - 1$.

Variable-Byte Coding: In variable-byte coding, an integer n is compressed as a sequence of bytes. In each byte, we use the lower 7 bits to store part of the binary representation of n, and the highest bit as a flag to indicate if the next byte is still part of the current number. Variable-byte coding uses $\lfloor \log_{128}(n) \rfloor + 1$ bytes to represent a number n; for example, $n = 267 = 2 \cdot 128 + 11$ is represented by the two bytes 10000010 00001011. Variable-byte coding is simple to implement and known to be significantly faster than traditional bit-oriented methods such as Golomb, Rice, Gamma, and Delta coding [26]. In particular, [21] showed that variable-byte coding results in significantly faster query evaluation than those previous methods, which were CPU limited due to their large decompression costs.

The disadvantage of variable-byte coding is that it does not achieve the same reduction in index size as bit-aligned methods. The main reason is that even for very small integers, at least one byte is needed; this puts variable-byte coding at a disadvantage when compressing frequencies, or docIDs in very long lists where the differences between consecutive docIDs are small. Variations such as nibble-oriented coding have been proposed [21], but with only slight additional benefits.

Simple9 (S9) Coding: This is a recent algorithm proposed in [1] that achieves much better compression than variable-byte coding while also giving a slight improvement in decompression speed. Simple9 is not byte-aligned, but can be seen as combining word alignment and bit alignment. The basic idea is to try to pack as many integers as possible into one 32-bit word. To do this, Simple9 divides each word into 4 status bits and 28 data bits, where the data bits can be divided up 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).

Overall, Simple9 has nine different ways of dividing 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 number. We use the four status bits to store which of the 9 cases is used. Decompression can be done in a highly efficient manner by doing a switch operation on the status bits, where each of the 9 cases applies a fixed bit mask to extract the integers.

Simple16 (S16) Coding: Simple9 wastes bits in two ways, by having only 9 cases instead of the 16 that can be expressed with 4 status bits, and by having unused bits in several of these cases. This motivated us to come up with a new variation that avoids this. Consider for example the case of 5 5-bit numbers in Simple9, with 3 bits unused. We can replace this with two new cases, 3 6-bit numbers followed by 2 5-bit numbers, and 2 5-bit numbers followed by 3 6-bit numbers. Thus, if four of the five next values are less than 32, and the other one less than 64, then at least one of these two cases is applicable, while Simple9 would we able to fit only four of the numbers into the next word using 4 7-bit numbers. Overall, we identified several such cases, including cases such as 10 2-bit numbers followed by 8 1-bit numbers and vice versa, for a total of 16 cases where each case uses all bits. This can again be implemented using a switch statement and fixed bit masks for each case.

We note here that [1] also proposed two methods called *Relate10* and *Carryover12* that in some cases achieve slight improvements over Simple9. We found that Simple16 fairly consistently outperformed these two methods, though some of their ideas could potentially also be added to Simple16. We also experimented with a number of variations of Simple16 that select the 16 cases differently; while some of these obtained improvements for the case of small values (i.e, frequency values, or docIDs in very long lists), overall the extra benefits were limited (up to 5% additional size reduction if we select the best settings for each list). Recent work in [2] also proposed another variation of Carryover12 called *Slide*.

PForDelta Coding: Our next method is a very recent technique proposed in [11, 27] for compression in database and IR systems. It is part of a larger family of compression schemes, and here we only describe and adapt the version that appears most suitable for inverted lists. PForDelta is neither word-aligned nor byte-aligned. It compresses integers in batches of multiples of 32. For example, for our setup, we compress the next 128 integers in one batch.

To do so, we first determine the smallest b such that most (say at least 90%) of the 128 values are less than 2^b. We then code the 128 values by allocating 128 b-bit slots, plus some extra space at the end that is used for the few values that do not fit into b bits, called *exceptions*. Each value smaller than 2^b is moved into its corresponding b-bit slot. We use the unused b-bit slots of the exceptions to construct a linked list, such that the b-bit slot of one exception stores the offset to the next exception (i.e., we store x if the current exception is the i-th value and the next exception is the $(i+x+1)$-th value). In the case where two exceptions are more than 2^b slots apart, we force additional exceptions in between the two slots. We then store the actual values of the exceptions after the 128 b-bit slots. In [11, 27], 32 bits are used for each exception; instead, we use either 8, 16, or 32 bits depending on the largest value in the batch. Finally, we also store the location of the first exception (the start of the linked list), the value b, and whether we used 8, 16, or 32 bits for the exception.

Each batch of integers is decompressed in two phases. In the first phase, we copy the 128 b-bit values from the slots into an integer array; this operation can be highly optimized by using a hardcoded bit-copy procedure for each value of b. That is, as suggested in [11], we have an array of functions $f[0]()$ to $f[31]()$, where $f[i]()$ is optimized to copy $(i+1)$-bit values in multiples of 32, and call the right function based on the value of b. Note that for any value of b, we have word-alignment at least after every 32 slots, allowing efficient hardcoding of bit masks for copying a set of 32 b-bit numbers. (This is the reason for using batches of size a multiple of 32.) In the second phase, called *patch phase*, we walk the linked list of exceptions and copy their values into the corresponding array slots.

The first phase of decompression is extremely fast, because it involves a hard-coded unrolled loop, with no branches that could be mispredicted by the processor. The second phase is much slower per element, but it only applies to a relatively small number of exceptions. This is in fact the main insight behind the PForDelta method, that significantly faster decompression can be achieved by avoiding branches and conditions during decompression. In contrast, while variable-byte coding is conceptually very simple, it involves one or more decisions per decoded integer, as we need to test the leading bit of each byte to check if another byte is following. This limits the performance of variable-byte coding on current processors. Simple9 has to perform one branch for each compressed word, i.e., typically every 6 to 7 numbers that are decoded.

We made two changes in our implementation compared to the description in [11, 27]. We use 8 or 16 or 32 bits per exception instead of always 32, and we allow any value of b while [11, 27] prefers to use a minimum value of $b = 8$. Each of these changes resulted in significant decreases in compressed size while we did not observe significant decreases in decompression speed. The results in [11, 27] show a larger compressed size than variable-byte coding for PForDelta; our results will show a much smaller compressed size, comparable to Simple9 (see Figures 4 and 7).

Rice Coding: Rice Coding is one of the oldest bit-aligned methods. It performs well in terms of compressed size, but is usually fairly slow compared to more recent methods. Our goal here was to try to challenge this assumption, by developing a high-performance implementation based on some new tricks.

We first describe Golomb coding, where an integer n is encoded in two parts: a quotient q stored as a unary code, and a remainder r stored in binary form. To encode a set of integers, we first choose a parameter b; a good choice is $b = 0.69 \cdot ave$ where ave is the average of the values to be coded [26]. Then for a number n we compute $q = \lfloor n/b \rfloor$ and $r = n \bmod b$. If b is a power of two, then $log(b)$ bits are used two store the remainder r; otherwise, either $\lfloor log(b) \rfloor$ or $\lceil log(b) \rceil$ bits are used, depending on r.

Rice coding is a variant of Golomb coding where b is restricted to powers of two. The advantage is that the number of bits to store r is fixed to be $log(b)$, allowing for a simpler and more efficient implementation through the use of bit shifts and bit masks. A possible disadvantage is that the compressed data may require more space; in practice the difference is usually very small.

Despite the restriction to powers of two, Rice coding is often significantly slower than variable-byte coding (by about a factor of 4 to 5). This is primarily due to the leading unary term as we need to examine this term one bit at a time during decompression. Influenced by the PForDelta method, we designed a new implementation of Rice coding that is significantly faster than the standard approach as follows: As in PForDelta, we code 128 (or some other multiple of 32) numbers at a time. We first store all the binary parts of the Rice code, using $log(b)$ bits each, in a $(128 \cdot b)$-bit field. Then we store the unary parts. During decompression, we first retrieve all the binary components, using the same optimized codes for copying fixed bit fields used in PForDelta. Then we parse the unary parts and adjust the decoded binary values accordingly. This is done not one, but eight bits at a time, thus processing the unary codes belonging to several numbers in a single step. In particular, we perform a switch on these eight bits, with 256 cases, where each case hardcodes the resulting corrections to the numbers. (The code for this consists of about 1500 lines generated by a script.)

This approach indicates an interesting relationship between Rice coding and PForDelta: Essentially, PForDelta chooses a value of b large enough so that almost all numbers consist of only a binary part, and then codes the few exceptions in a fairly sloppy but fast way. Using the best choice for b in Rice coding, many of the numbers are larger than b and need to be adjusted based on the unary part of the code; thus we have to be more careful on how to code these cases (i.e., we cannot simply use 32 bits each or use a linked list to

identify such numbers). Note that by choosing b slightly larger in our Rice code, we can get more speedup at a slight increase in size.

Implementation: All methods were implemented and optimized in C^{++}, taking into account the properties of current CPUs. The code is available at `http://cis.poly.edu/westlab/`. Some of the methods, in particular PForDelta and our implementation of Rice Coding, are not suitable for very short lists, as they operate on multiples of 32 or 128 numbers. To address this issue, we always used variable-byte coding for any inverted list with fewer than 100 postings, while for longer lists we padded the inverted lists with dummy entries as needed by the various compression methods. We note that such short lists make up more than 98% of all inverted lists, but that they contain less than 0.1% of all index postings. Thus, our decision to use variable-byte on short lists did not have a measurable impact on query processing performance. (However, for Rice and PForDelta, padding very short lists to 128 postings would increase index size by several percent.)

4.2 Experimental Setup

Before presenting our experimental results, we describe our setup, which we also use in later sections. The data set we used in our experiments is a set of 7.39 million web pages selected randomly from a crawl of 120 million pages crawled by the PolyBot web crawler [22] in October of 2002. The total compressed size was 36 GB (using `gzip` on files of about 100 pages each), and the total number of word occurrences was 5,868,439,426 (about 794 words per document). After parsing the collection, we obtained 1,990,220,393 postings (about 269 distinct words per document), and there were a total of 20,683,920 distinct words in the collection. To evaluate our compression algorithms and caching policies, we generated several sets of inverted index structures using different compression methods. All our experiments are run on a machine with a single 3.2GHz Pentium 4 CPU and 4 GB of memory. For caching, we chose a block size of 64 KB as the basic unit. We use this experimental setup throughout this paper.

Our query trace was taken from a large log of queries issued to the Excite search engine from 9:00 to 16:59 PST on December 20, 1999. We removed 50 stopwords in the queries and eliminated any duplicate queries as (most of) these would usually be handled by a result caching mechanism. This gave us 1,135,469 unique queries where each query has 3.43 words on average. Of these, we use 1,035,469 to warm up our cache, and the remaining 100,000 to measure the performance of our compression and caching algorithms.

In addition, we also experimented with two sets of queries extracted from a recent trace of over 36 million queries from about 650,000 AOL users. We processed the trace in two ways such that the number of unique queries is equal to that of the Excite trace, by limiting the time frame and by limiting the number of users considered, in order to get a similar trace length to that of the processed Excite trace. Even though this trace is much more recent, we got very similar statistics for the lengths of queries and accessed inverted lists. Moreover, when we ran our experiments on all three sets, we obtained very similar numbers for the old Excite trace and the much newer AOL data. Thus, we omit figures for the AOL data.

We comment briefly about our removal of duplicate queries. We considered queries containing the same words in a different order to be duplicates, which might decrease the hit ratio of our list caching mechanism. Many current engines have to take word order as well as maybe other features such as user location into account during result caching. Also, by removing all duplicates we assume that the result cache is large enough to cache all queries in the trace. Note that our only goal here is to obtain query traces with realistic properties; on average queries after result caching have more terms than the queries submitted by users, as shown in Figure 3, since shorter queries are more likely to be in the result cache. We are not concerned with details of result caching such as cache size and eviction policy, which should only have a minor impact on our experiments.

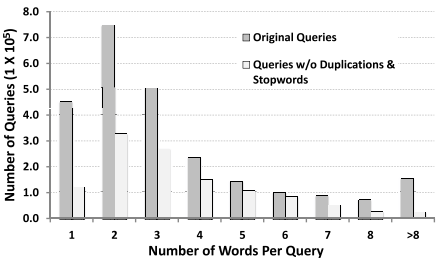

Figure 3: Number of queries before and after removing duplicate queries and stopwords in queries.

4.3 Comparison of Compression Algorithms

We now present our results from the experimental evaluation of the different compression algorithms. We focus on two performance measures, the sizes of the compressed index structures and the time it takes to decompress them. We are not overly concerned with compression speed, as this is a one-time operation while decompression happens constantly during query processing. All methods allow for reasonably fast compression, at least at rates of tens of millions of postings per second, but we did not optimize this.

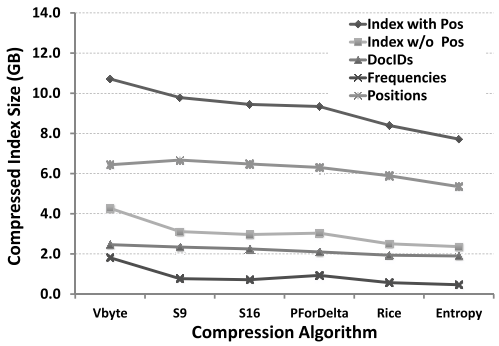

Figure 4: Total index size under different compression schemes. Results are shown for docIDs, frequencies, positions, an index with docIDs and positions only, and an index with all three fields.

We first look at the resulting size of the entire index structure for the 7.39 million pages, shown in Figure 4. In this figure, we add one additional method called *entropy* that uses the global frequency distribution of the compressed integers to provide a naive bound on possible compression (assuming random assignment of docIDs and no clustering of word occurrences within documents). We see that Rice coding performs best, and that variable-byte performs the worst in nearly all cases, with the exception of position data where S9 is slightly worse and S16 and PForDelta are about the same. For frequencies, variable-byte results in a size about 2 to 3 times larger than the other methods, while for docIDs the differences are more modest. We note that an index structure with positions is typically 2.5 to 3 times larger than one without, since when a word occurs in a document there are on average about 3 occurrences of that word. We now look at the results in more detail.

Figure 5 shows how the compression ratio for docIDs depends on the lengths of the inverted lists. (Here, we divide the range of list lengths between the shortest and longest list into five intervals of equal size to define the 5 groups of lengths.) When lists are very short, the gaps between the docIDs are large and consequently more bits per docID are needed. As lists get longer, compression improves significantly. However, improvements for variable-byte are limited by the fact that at least one byte is needed for each com-

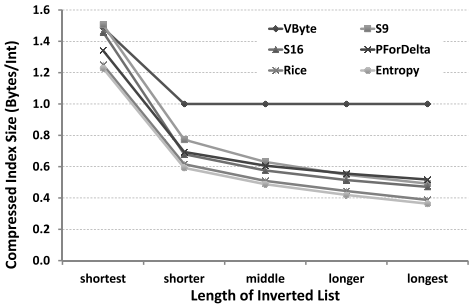

Figure 5: Comparison of algorithms for compressing DocIDs on inverted lists with different lengths.

pressed number, while the other methods use only around 4 bits per docID on the longest lists. For frequencies (not shown due to space constraints), the numbers to be compressed increase with the lengths of the lists, since there is a higher chance of a term occurring several times in a document. In this case, variable-byte almost always uses one byte per frequency value, while the other methods usually use less than 4 bits, with slight increases as lists get longer. The values of the positions (also not shown) tend to be much larger than the docID and frequency values; thus variable-byte is less disadvantaged by its restriction to byte boundaries, and overall the differences between the methods are much smaller, as we saw in Figure 4.

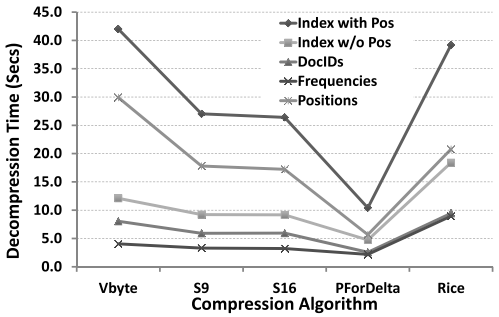

Figure 6: Times for decompressing the entire inverted index.

Next, we look at the decompression speed, which is extremely important as it impacts the speed of query processing. In Figure 6, we show the times for decompressing the entire inverted index structure using our methods. (The numbers assume that the index data is already in main memory, and are based on total elapsed time on a dedicated machine.) While Rice coding obtains the best compression ratio among the realistic algorithms (i.e., excluding entropy), it is slowest when decompressing docIDs and frequencies. Note that variable-byte, which was worst in terms of compression ratio, is the second-slowest method on docIDs and frequencies, and even slower than Rice on position data. The poor speed of variable-byte on position data is primarily due to the fact that position values are larger and more often require 2 bytes under variable-byte; this case tends to be much slower due to a branch mispredict. We also note that PForDelta is the fastest of all the methods on all three data fields, by a significant margin. S9 and S16 take about twice as long, but are still faster than Rice and variable-byte.

These results are based on compressing and decompressing the entire index. However, not every inverted list is equally likely to be accessed during query processing. Certain terms occur more frequently in user queries, and the likelihood of a term occurring in queries is often not related to that of occurring in the collection. To investigate the performance of the compression methods during query processing, we used the last 100,000 queries of the trace.

In Figure 7, we look at the average amount of compressed data that is accessed for each query. Here, we assume that the com-

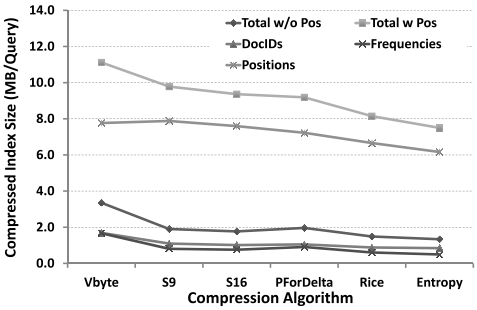

Figure 7: Average compressed size of the data required for processing a query from the Excite trace.

plete inverted lists of all terms are decompressed during execution of a query, without skips or early termination during index traversal. (Otherwise, the total cost would be lower though we would expect the same relative behavior of the different methods.) While the overall picture is similar to that in Figure 4, there are also a few differences. In particular, we find that the amount of data that is accessed per query is now more than 4 times larger for an index with positions than for one without, while the difference in overall index size was only a factor of 2.5 to 3. This is due to the fact that longer lists are used more frequently than shorter lists during query processing, and such longer lists also tend to have multiple occurrences of the word in a single document. For the same reason the difference in compressed size between docIDs and frequencies is now much smaller. Finally, we note that while the graph for docIDs may seem almost flat in the figure, this is not quite the case; the total docID cost per query decreases from 1.69 MB for variable-byte to about 1.05 MB for PForDelta, a reduction of almost 38%.

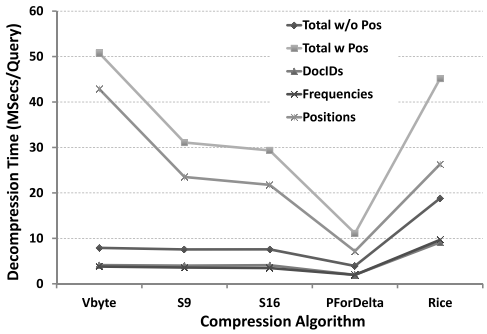

Figure 8: Average decompression time (CPU only) per query, based on 100,000 queries from the Excite trace.

Figure 8 shows the average time for decompressing all inverted lists accessed by a query, based on 100,000 queries from the Excite trace. Overall, the relative ordering of the methods is similar to that for the entire index in Figure 6.[2] For docIDs and frequencies, the times of variable-byte, S9, and S16 are similar, while PForDelta is about twice as fast and Rice coding is two times slower. For position data, however, Rice is slightly faster than variable-byte, while PForDelta is about three times faster than S9 and S16 and five times faster than variable-byte.

Algorithm	Total w/o Pos	Total w Pos	DocID	Freq	Pos
VByte	441.99	174.55	424.68	460.78	125.26
S9	461.36	285.13	437.29	488.25	228.39
S16	460.75	301.91	425.47	502.41	246.62
PForDelta	889.14	798.38	889.69	888.59	748.68
Rice	185.60	196.30	191.23	180.30	203.95

Table 1: Decompression speeds in millions of integers per second.

[2]Note that the cost of decompressing the entire index is similar to that for about 1000 queries, since many of the query terms are fairly common words.

Finally, in Table 1 we show the decompression speeds of the different algorithms in millions of integers per second. We see that PForDelta not only has the highest speed, but its speed is also not much affected by the average values of the integers that are decoded. In contrast, variable-byte and to a lesser extend S9 and S16 become significantly slower for larger integer values such as position values. Rice coding is also not much affected by the values. In summary, it appears that variable-byte coding does not perform well when compared to other recent techniques, which outperform it in terms of both compression ratio and speed, and that PForDelta in particular performs very well overall.

Finally, we also studied optimizations for frequency data. An important difference between docIDs and frequencies (or positions) is that a frequency value only needs to be retrieved when a score is computed. Also, decompressing a value does not necessarily require decompression of all previous values in the chunk. If query processing is based on intersection of the query terms, then only relatively few frequencies need to be retrieved. This allows for a variety of optimizations in organizing frequency data. One simple approach combines pairs of consecutive frequencies into a single value by "shuffling" their bits (i.e., "000" followed by "111" becomes "010101"), and then applies PForDelta to the resulting 64 pairs in each chunk. This resulted in about 10% better compression and almost twice as fast decompression. Of course, to actually use the frequency, we have to select the relevant bits from the uncompressed number, but this is fast if applied to only a few of the postings (results omitted due to space limits).

5. LIST CACHING POLICIES

We now consider list caching policies. Search engines typically use a significant amount of the available main memory, from several hundred megabytes to multiple gigabytes per machine, to keep the most frequently accessed parts of the inverted index in main memory and thus reduce data transfers from disk during query processing. Although modern operating systems also perform caching of disk data, search engines tend to employ their own mechanisms and policies for better performance.

In our implementation, the inverted index consists of 64KB blocks of compressed data, and the inverted list for a word is stored in one or more of these blocks. Inverted lists usually do not start or end at block boundaries. We use these blocks as our basic unit of caching, and our goal is to minimize the number of blocks that are fetched from disk during query processing. We note here that there are at least three different objective functions that have been used for list caching in search engines: (a) *query-oriented*, where we count a cache hit whenever all inverted lists for a query are in cache, and a miss otherwise, (b) *list-oriented*, where we count a cache hit for each inverted list needed for a query that is found in cache, and (c) *block- or data size-oriented*, where we count the number of blocks or amount of data served from cache versus fetched from disk during query processing.

The query-oriented objective was recently studied in [3] and is more appropriate for architectures where a node holding only a cache of inverted lists can evaluate a query locally if all lists are in its cache, and otherwise has to forward it to another machine or backend search engine cluster. (Thus, the goal is to minimize the number of forwarded queries.) The list-oriented objective is appropriate when the cache is on the same machine as the index and we want to minimize the number of disk seeks. This is suitable for collections of moderate size, where the cost of fetching a list is dominated by the disk seek. In our experiments, we use the third objective, which we feel is more appropriate for very large collections of tens to hundreds of millions of pages per node (and thus very long inverted lists), where the goal is to minimize overall disk traffic in terms of bandwidth. Thus our results here cannot be directly compared to the recent work in [3] based on the query-oriented approach. In a nutshell, the main difference is that good policies for the first two objectives should give strong preference to keeping short lists in cache, while in our case this is not the case. One similarity with [3, 6] is that we also observe that fairly static methods (LFU with a long history in our case vs. static assignment of lists to the cache in their work) perform well as the term frequencies in available query logs do not appear to change enough over time to significantly impact caching performance.

Note that even though we do caching based on 64 KB blocks, all blocks of one inverted list are kept sequentially on disk, but are usually not kept sequentially in cache. Also, while we treat the blocks of an inverted list as separate entities, it usually happens that all blocks of a list are fetched together on a cache miss, and are also evicted at (around) the same time under all our caching policies (ignoring boundary blocks that also contain fragments of other lists). We use blocks here to avoid having to deal with the issue of cache space fragmentation that would arise if we cache individual lists of variable size. This means that we cache an entire 64 KB block in order to access a short list with only a few postings. However, for large collections most inverted lists that are accessed extend over multiple blocks, and thus this is not such a major issue. We now describe the caching policies that we study.

5.1 Policies for List Caching

Least Recently Used (LRU): This is one of the most widely used caching policies, and often used as a baseline to compare against. LRU discards the least recently used block first when the cache is full and a new block needs to be loaded into cache. To do so, LRU keeps track of the usage order of blocks in cache.

Least Frequently Used (LFU): LFU counts how often a block was accessed during some limited time period, say since it was last placed into cache or over some longer time. We evict the block with the lowest frequency of use. Thus, we need to keep track of the usage frequency of each block in cache, and possibly also some blocks currently out of cache if we keep track for a longer period of time. The performance of LFU depends on the length of the history that we keep: Increasing the period of time over which we track the frequency of a block may increase performance when the query distribution is fairly stable, but makes it more difficult for the cache to adapt to fast changes in the query distribution. We experimented with several settings for the length of the history. Note that the space of the statistics is not much of a problem in our case, since each block is fairly large. We also explored some variations of LFU, in particular a weighted LFU policy where we give higher importance to recent queries when computing the usage frequency. However, despite trying various weighting schemes we did not see any significant performance over basic LFU.

Optimized Landlord (LD): *Landlord* is a class of algorithms for weighted caching that was proposed and studied in [9, 24], and recently applied to another caching problem in search engines in [14]. When an object is inserted into the cache, it is assigned a deadline given by the ratio between its benefit and its size. When we evict an object, we choose the one with the smallest deadline d_{min} and also deduct d_{min} from the deadlines of all remained objects. (This can be implemented more efficiently by summing up everything that has been deducted in a single counter, rather than deducting from every deadline.) Whenever an object already in cache is accessed, its deadline is reset to some suitable value. In our case, every object has the same size and benefit since we have an unweighted caching problem. In this case, if we reset each deadline back to the same original value upon access, Landlord becomes identical to LRU.

However, when we follow a different rule for resetting the dead-

lines, we obtain a new class of policies that is in fact quite useful for certain unweighted caching problems. This set of policies was first proposed in [14], and we call it *Optimized Landlord*. We use the following two modifications: First, in order to give a boost to blocks that have already proven useful, we give longer deadlines to blocks that are being reused compared to blocks that just entered the cache. In particular, a renewed block gets its original deadline (upon insertion) plus a fraction α of its remaining (unused) deadline. We use $\alpha = 0.5$, which works well on our data as well as the one in [14]. Second, we use a cache admission policy, as suggested in [14], and only insert a block into cache if it was previously used at least once during some suitably chosen time window. This avoids inserting objects that are unlikely to be accessed ever again due to the highly skewed distribution of terms in queries. (This issue was also recently addressed in [4] for the case of result caching.)

Multi-Queue (MQ): MQ is studied in [25]. The idea is to use not one but m (say, $m = 8$) LRU queues $Q_0, Q_1, \ldots, Q_{m-1}$, where Q_i contains pages that have been seen at least 2^i times but no more than $2^{i+1} - 1$ times recently or that have been seen at least 2^{i+1} times but have been evicted from queues at a higher level. The algorithm also maintains a history buffer Q_{out}, which keeps the frequency information about pages that were recently ejected. On a cache hit, the frequency of the block is incremented and the block is placed at the Most-Recently-Used (MRU) position of the appropriate queue, and its *expireTime* is set to *currentTime* + *lifeTime*, where *lifeTime* is a tunable parameter. On each access, the *expireTime* for the LRU position in each queue Q_i with $i > 0$ is checked, and if it is less than *currentTime*, then the block is moved to the MRU position of the next lower queue. When a new block is inserted into a full cache, the LRU block in the lowest nonempty queue is evicted and its frequency information is put into Q_{out}.

Adaptive Replacement Cache (ARC): This policy was proposed in [16, 17], and like Landlord and MQ also tries to balance recency (LRU) and frequency (LFU). Conceptually, ARC operates on two levels: it maintains (i) two lists L_1 and L_2 referring to c pages each that were recently accessed but that may or may not be in cache, and (ii) the actual cache of size c. L_1 keeps track of pages that have been recently seen only once, while L_2 keeps track of pages that have been seen at least twice. Thus, L_1 and L_2 together keep track of $2c$ pages of which c are in cache while the others were recently evicted. Of the pages actually in the cache, about p are from L_1 and $c - p$ from L_2, where $p \in [0, \ldots c]$ is a parameter tuned in response to the observed workload. Eviction from the actual cache is then done according to a (slightly involved) set of rules that takes L_1, L_2, and p into account; see [16, 17] for details.

5.2 Comparison of List Caching Policies

In the following experiments, we use the same data and setup as in the previous section. All results are based on an index without positions compressed using PForDelta. (Overall behavior is very similar for an index with positions, assuming cache size is scaled with increased total index size.) We always make sure to warm up the cache by first running over most of the query log, and then measuring only the performance of the last 100,000 queries. Before comparing all methods, we first run experiments to tune LFU with respect to the history size that is kept.

Figure 9 compares the cache hit ratios of LFU with different history lengths. In particular, a history of *k x Cache Size* means that we can keep frequency statistics for k times as many blocks as fit into cache. For $k = 1$, we keep only stats about pages that are in cache, while for larger k the history itself is also maintained using LFU. For small cache sizes (up to 15% of the total index size), we observe a measurable benefit (up to 4% higher hit rate) from having a longer history. However, a value of $k = 5$ suffices to get almost

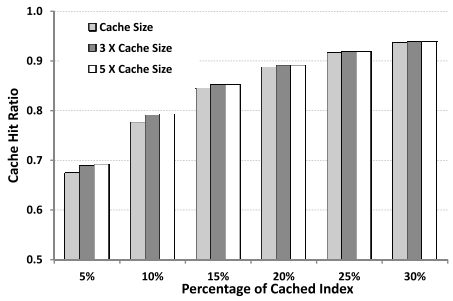

Figure 9: Impact of history size on the performance of LFU, using an index compressed with PForDelta.

all the possible benefit. One drawback of a longer history is the space overhead, but this is small since we only need a few bytes for each 64 KB block. The impact of history size is negligible when more than 25% percent of the index fits in cache.

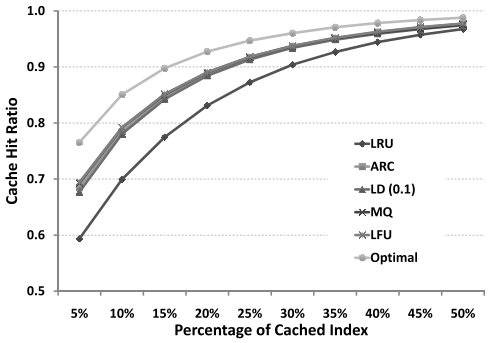

Figure 10: Cache hit rates for different caching policies and relative cache sizes. The graphs for LFU and MQ are basically on top of each other and largely indistinguishable. *Optimal* is the clairvoyant algorithm that knows all future accesses and thus provides an upper bound on hit ratio. Each data point is based on a complete run over our query trace, where we measure the hit ratio for the last 100,000 queries.

After tuning LFU, we now compare all algorithms in terms of hit rate; results are shown in Figure 10. We see that LRU performs consistently worst, while LFU and MQ perform best. Both Landlord and ARC also come close to the best methods. Note that the improvements over LRU are quite significant, with hit rates increasing by up to 10% in absolute terms (for small cache size), or conversely with miss rates and thus total disk traffic decreased by up to a third.

Note that we plot on the x-axis not the absolute size of the cache, but the percentage of the total index that fits in cache. Figure 10 was obtained on 7.36 million pages using PForDelta for list compression, but we found that results were basically identical for other compression methods and other (sufficiently large) collections sizes. That is, the cache hit rate depends primarily on the caching policy and the percentage of the index that fits into cache, and not on compression method and absolute index size.

5.3 Burstiness, Static Caching, and a Hybrid

In our experiments, LFU at least slightly outperformed all other methods. While LFU is a natural fit for traces where term frequencies are fairly stable over longer periods of time, it is not good at exploiting any local bursts where a particular item occurs repeatedly over a short period of time. Such burstiness is known to be crucial for the performance of result caching policies [15, 23], motivating hybrid schemes such as SDC in [10]. We ran several experiments to evaluate whether we could get an improvement over LFU by exploiting burstiness. An analysis of the gaps between occurrences of terms in the trace showed almost no burstiness. An attempt to design a hybrid scheme that uses the total number of occurrences

as well as the distance to the last occurrence and the gaps between recent occurrences led to only tiny improvements (details omitted).

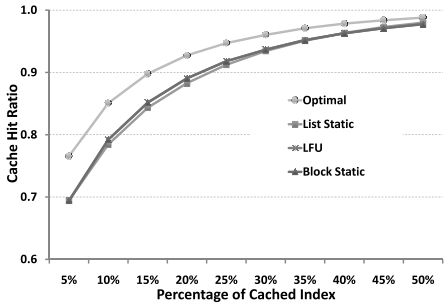

Figure 11: LFU, static block-based, and static list-based caching.

This seems to agree with recent work in [3] that suggests using a static assignment of lists to the cache (though for a different caching model). Thus, we ran another experiment that compares LFU to two versions of a simple static scheme adapted from [3]: one that uses blocks (as our LFU) and one that uses lists. (One advantage of a static scheme is that cache fragmentation is not an issue, making it more attractive to use lists rather than blocks.) We used the first 1,035,469 queries in our trace to compute a static cache assignment, and the last 100,000 queries to test the assignment. The results are shown in Figure 11, which indicates that the performance of LFU is essentially identical to a static assignment. Using a list-based static assignment results in a very slightly lower hit ratio, but there is a catch: In our block-based schemes, slightly more data will be fetched at the same hit ratio, since we fetch an entire block whenever some part of it is needed for query processing. Overall, all three schemes result in almost the same amount of disk traffic. Note however that a static approach may not be suitable in scenarios where query distribution changes periodically due to, e.g., users from different countries and time zones doing searches in different languages on the same engine.

Finally, we also experimented with a hybrid cost model and algorithms where both disk seeks and data transfer times are taken into account; we found that meaningful performance improvements (up to 12% reduction in disk access time) can be obtained for our 7.39 million document collection by modifying LFU to take seek times into account, particularly for large cache sizes.

6. COMPRESSION PLUS CACHING

In previous sections, we separately discussed and evaluated list compression and list caching techniques. As we found, there are significant differences between the methods, and also the choice of the best caching policy does not depend on the choice of compression. However, this does not mean that the performance of caching is independent of compression. In real systems, for a given collection size we have a fixed absolute amount of memory that can be used for caching. Thus, a better compression method means that a larger fraction of the total index fits in cache, and thus a higher cache hit ratio should result. In this section, we study the impact of combining index compression and index caching, which does not seem to have been previously addressed. All experiments in this section are run on an index without positions using the complete query trace, but measuring only the last 100,000 queries.

6.1 Evaluation of the Combination

Figure 12 compares the performance of the LFU caching policy with different compression methods applied to the index. In this case, the memory size is in MB, and thus a better compression method results in a higher percentage of the index in cache. Consequently, we see that variable-byte achieves the worst cache hit rate, while Rice and the theoretical Entropy achieves the best. However, S9, S16, and PForDelta are also significantly better than variable-byte, and quite close to the optimal. Overall, absolute hit rates are improved by about 3 to 5%. In fact, for a cache size of 1280 MB, the cache miss rate, and thus total disk traffic, is almost cut in half by switching from variable-byte to Rice coding. Thus, index compression methods have an important extra benefit in caching when disk bandwidth is a performance bottleneck.

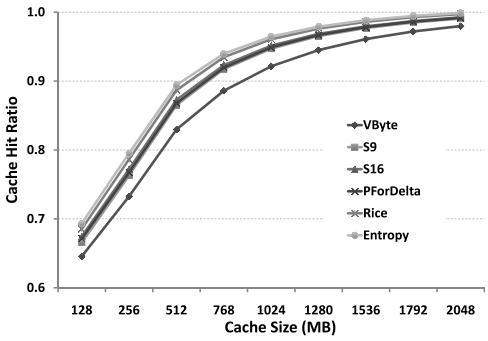

Figure 12: Comparison of compression algorithms with different memory size, using 100,000 queries and LFU for caching. Total index size varied between 4.3 GB for variable-byte and 2.8 GB for Entropy.

Next, we try to analyze the combined performance of compression and caching using a simple performance model. Here, we assume that the cost of a query consists of the time for disk access (if any) and the time spent on decompression of the inverted lists retrieved from either disk or cache. We note that this is a somewhat naive model, as it assumes that inverted lists are fetched from disk in their entirety (no early termination), that they are decompressed in their entirety (no skipping over compressed chunks), and that other costs such as score computation are minimal. These assumptions are not really realistic for large engines, but we could modify the model to account for the percentage of each list that is actual decompressed; the most suitable choice of modifications however depends on various details of query execution and ranking function and thus we stick to our simple model. We believe that it does provide a useful benchmark of the lower search engine layer involving index fetch and decompression. We also limit ourselves to three methods, PForDelta, S16, and Rice coding, since the other two methods are strictly dominated by these three.

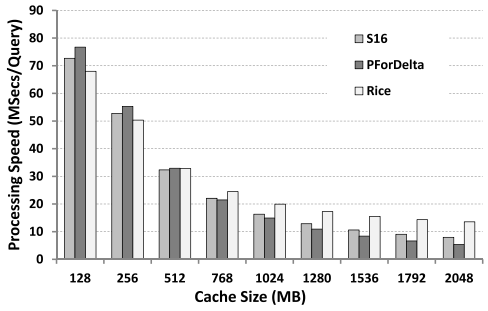

Figure 13: Comparison of PForDelta, S16, and Rice coding, using LFU caching and assuming 10MB/s disk speed.

Not surprisingly, as shown in Figure 13, query processing costs decrease significantly with larger cache sizes when disk is slow. In this case, Rice coding performs better than S16 and PForDelta on small cache sizes, where its better compression ratio translates into a higher cache hit rate. For larger cache sizes, disk is less of a factor and thus PForDelta substantially outperforms the other two methods. The situation changes somewhat for faster disks (50 MB/s), as shown in Figure 14. Here, PForDelta dominates for the entire range

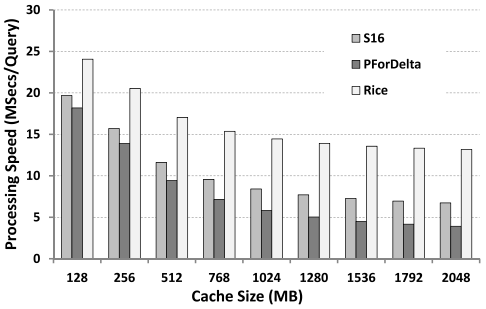

Figure 14: Comparison of PForDelta, S16, and Rice coding, using LFU caching and assuming 50MB/s disk speed.

of cache sizes, followed by S16 and then by Rice coding. The reason is that the slightly higher cache hit rates for S16 and Rice do not make up for the much slower decompression.

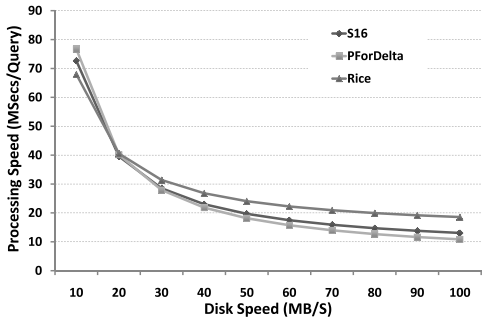

Figure 15: Comparison of query processing costs for different disk speeds, using LFU and a 128MB cache.

Finally, we look at the impact of disk speed for a fixed cache size. In Figure 15, we show results for a cache size of 128 MB and disk speeds between 10 to 100 MB/s. We see that when the disk has a transfer rate up to 20 MB/s, Rice is faster than the other compression algorithms, but otherwise PForDelta is best. (To be precise, S16 is briefly the champion around 20 MB/s, by a very slim margin.) Looking at disk speeds of 20 MB/s or less may seem like a useless exercise given that current cheap disks already achieve transfer rates of about 60 MB/s, but we note that our results are really based on the ratio between disk transfer rate and decompression speed: If CPUs increase in speed by a factor of 2 this would have the same effect in relative terms as disk speeds being reduced by a factor of 2. Thus, architectural trends in the future may make techniques such as Rice coding relevant again. For current architectures, PForDelta plus LFU appears to be the best choice.

7. CONCLUDING REMARKS

In this paper, we studied techniques for inverted index compression and index caching in search engines. We provided a detailed evaluation of several state-of-the-art compression methods and of different list caching policies. Finally, we looked at the performance benefits of combining compression and caching, and explored how this benefit depends on machine parameters such as disk transfer rate, main memory, and CPU speed.

There are several interesting remaining open problems. First, the recent work on PForDelta in [11, 27] shows that decompression speed depends not so much on questions of bit versus byte alignment, but on the amount of obliviousness in the control and data flow of the algorithm. Any approach that requires a decision to be made for each decoded integer will be at a severe disadvantage in terms of speed. We have shown here that ideas similar to PForDelta can also be used to increase the speed of Rice decoding, implying a trade-off between speed and compression. An interesting challenge is to design other methods that improve on PForDelta in terms of compression while matching (or coming close to) its speed.

Another interesting open question concerns the compression of position data. As we saw in our experiments, an index with position information requires about 4 to 5 times as much data to be traversed per query than an index without positions, while the amount of data fetched from disk is often an order of magnitude higher since a much smaller fraction of such an index fits in cache. There are a number of compression methods (see, e.g., [26]) that can significantly improve compression over the simple methods considered here. However, these methods tend to be fairly slow, and it is also not clear how to best apply them to position information in web page collections. We plan to address this in future work.

8. REFERENCES

[1] V. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. of the 15th Int. Australasian Database Conference*, pages 61–67, 2004.

[2] V. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. on Knowledge and Data Engineering*, 18(6), 2006.

[3] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th Annual SIGIR Conf. on Research and Development in Inf. Retrieval*, 2007.

[4] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. Witschel. Admission policies for caches of search engine results. In *Proc. of the 14th String Processing and Information Retrieval Symposium*, September 2007.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[6] R. Baeza-Yates and F. Saint-Jean. A three-level search-engine index based in query log distribution. In *Proc. of the 10th String Processing and Information Retrieval Symposium*, September 2003.

[7] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proc. of the Int. Symp. on String Processing and Information Retrieval*, pages 25–28, 2005.

[8] St. Büttcher and C. Clarke. Index compression is good, especially for random access. In *Proc. of the 16th ACM Conf. on Inf. and Knowledge Manag.*, 2007.

[9] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of the USENIX Symp. on Internet Technologies and Systems*, 1997.

[10] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. on Information Systems*, 24, 2006.

[11] S. Heman. Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 2005.

[12] B. Jonsson, M. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 118–129, June 1998.

[13] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. of the 12th Int. World-Wide Web Conference*, 2003.

[14] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of the 14th Int. World Wide Web Conference*, 2005.

[15] E. Markatos. On caching search engine query results. In *5th International Web Caching and Content Delivery Workshop*, May 2000.

[16] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of the USENIX Conf. on File and Storage Technologies*, 2003.

[17] N. Megiddo and D. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *IEEE Computer*, 37(4), 2004.

[18] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. on Information Systems*, 14(4):349–379, 1996.

[19] G. Navarro, E. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retrieval*, 3(1), 2000.

[20] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in Inf. Retrieval*, 2001.

[21] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval*, August 2002.

[22] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, 2002.

[23] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proc. of the Infocom Conference*, 2002.

[24] N. Young. On-line file caching. In *Proc. of the 9th Annual ACM-SIAM Symp. on Discrete algorithms*, 1998.

[25] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Annual Techn. Conf.*, 2001.

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

[27] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the Int. Conf. on Data Engineering*, 2006.