Efficient In-Memory, List-Based Text Inversion

David Hawking
Microsoft
Canberra, ACT, Australia
david.hawking@acm.org

Bodo Billerbeck
Microsoft
Melbourne, Victoria, Australia
bodob@microsoft.com

ABSTRACT

When building a large inverted file index on a system with effectively unlimited memory, performance may be constrained by RAM latency. To optimise speed requires an understanding of the non-uniform memory access characteristics of modern systems. We address three main techniques for improving the performance of an in-memory, list-based inverted file indexer: List chunking, in-chunk postings compression, and use of virtual memory "Large Pages". We compare performance of dynamic chunking schemes capable of adapting to the Zipf-like distribution of term frequencies. Using a data set with 8.5 billion word occurrences, we find that the techniques are cumulative. Chunking almost halves the memory required for linked lists, while dramatically reducing the number of cache-line reads required to traverse the lists; In-chunk compression further halves the memory footprint, though it does not make much difference to speed; Large pages reduce the inefficiency of page table walks and speed up both phases of index building.

KEYWORDS

Information retrieval, Inverted lists, Indexing

ACM Reference format:
David Hawking and Bodo Billerbeck. 2017. Efficient In-Memory, List-Based Text Inversion. In *Proceedings of The 22nd Australasian Document Computing Symposium, Brisbane, QLD, Australia, December 7–8, 2017 (ADCS 2017),* 8 pages.
https://doi.org/10.1145/3166072.3166080

1 INTRODUCTION

We analyse the performance of an inverted file text indexer which uses a vocabulary structure, plus in-memory linked lists to achieve the inversion. We show that the structure of the linked lists can have a significant effect on performance. The data sets we study comprise large numbers of quite short documents, for example web page and academic paper titles or web search query logs. These data sets are characteristic of the intended applications in retrieval, classification, and query suggestion. We assume that documents are ordered in descending order of a single static score. We also assume the *off-line* case in which new versions of an index are built off-line rather than the case where an index is updated on the fly. Our method is *one pass*, meaning that the input documents are scanned only once. We consider the indexing of text data sets containing many billions of term occurrences.

In contrast to most previous work on efficient indexing, we assume that the hardware configuration used for indexing has sufficient RAM to allow the inversion to occur entirely in memory. This is not an unrealistic assumption as servers with multiple terabytes of RAM are now readily (if not cheaply) available. Our focus is thus on in-memory optimisations. Most modern text retrieval systems use inverted indexes (concordances) which were first computerised by Taube [12]. An efficient lookup structure for indexable terms gives access to all occurrences of each term through lists of postings. Each posting describes an individual term occurrence and contains a payload which consists of the document number plus optional additional information, e.g. within document occurrence positions.

In a survey article, Zobel and Moffat [16, pp. 15–17] describe a range of different approaches to building inverted indexes. They claim that dynamically resizable arrays are the best choice for representing postings lists built up in memory. The chunked linked lists which we discuss in this paper can be seen as a flexible way of implementing dynamically resizable arrays. The BitFunnel indexing system [7] challenges the assumption that inverted files are superior to signature files in text indexing. Impressive performance is reported and BitFunnel easily handles incremental additions to the index, however, bit signatures seem less well suited to the very short documents in the data sets we study, and do not record term positions or term frequencies.

We study an approach in which postings lists are built up as documents are scanned, using a linked list structure similar to that of Harman and Candela [8]. However, their linked lists were kept on disk, while ours are entirely in RAM. In Section 4.5 of their book, Büttcher et al. [4] describe a more efficient chunked version of linked lists. We take a similar approach and compare multiple models of chunking, and the use of compression within chunks. We note that Büttcher et al. found that chunked linked lists performed better than an alternative implementation of dynamically resizable arrays based on `realloc`. Achieving optimal indexing performance in a large RAM environment requires understanding of the features of modern system architectures summarised in the next section.

1.1 Recap of hardware architecture

Our study addresses the non-uniformity in memory latency which is characteristic of these designs. Specific results will vary across different systems but we believe that the main conclusions and advice will generalise provided the following assumptions are met:

(1) A large difference between RAM and cache latency;
(2) Each memory access causing the reading or writing a whole cache line (typically 64 bytes) at a time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ADCS 2017, December 7–8, 2017, Brisbane, QLD, Australia
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-6391-4/17/12…$15.00
https://doi.org/10.1145/3166072.3166080

(3) Multiple levels of cache; Typically, the L1 cache is split into two halves, one for instructions and the other for data. In multi-core CPUs, often the L1 and L2 caches are local to each core, while L3 is shared by all the cores on a CPU chip.
(4) A virtual memory architecture implemented by a memory management unit (MMU) with support for large/huge pages as well as standard 4kB ones.
(5) A limited-capacity Translation Look-Aside Buffer (TLB) to reduce the need for walking large page tables.

Because we are aiming to eventually handle datasets with up to 100 billion documents and a trillion postings, we also assume a 64 bit architecture and operating system. The translation between virtual addresses in a program and physical addresses in RAM is ultimately controlled by a page table structure. Operating systems supporting 48-bit virtual addresses[1] typically use a four-level 512-ary page table tree. Fully populated, such a page table would have $1 + 512 + 512^2 + 512^3$ entries, approximately 135 million, and occupy about 1GB of memory. That is obviously too large to fit in cache. Accordingly, even for pages which are present in RAM, address translation may be quite slow due to the need to "walk the page table". Page table walking is hopefully avoided in the majority of cases by the Translation Look-aside Buffer (TLB), which mediates every memory access. It is a small associative cache which is able to supply address translations in one clock cycle. A high rate of TLB misses seriously degrades performance.

Another memory access issue which complicates the study of in-memory performance in multi-CPU architectures is the partitioning of available RAM across CPUs. Access to memory associated with a different CPU is slower than accessing local memory. To illustrate the degree of non-uniformity of access times, for a representative CPU, the best access latency in clock cycles for different levels of memory in the Intel Core i7 Xeon 5500 series is 4 for L1 cache, 10 for L2 cache, 40 for L3 cache, 60 for local DRAM and 100 for remote DRAM[2]. For further reading on hardware and operating systems, the reader is referred to Arpaci-Dusseau and Arpaci-Dusseau [2], Intel [9], AnandTech [1] and the ISCA conference[3]. We wrote a simple memory access benchmark to explore these effects, and report results in the next section.

1.2 Memory access benchmarking

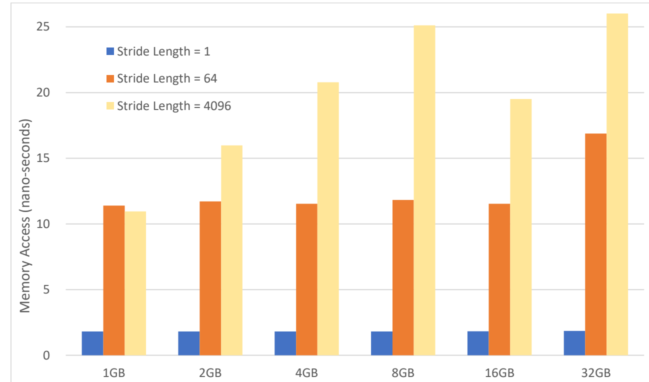
The memory access benchmark allocates a large block of memory and then accesses a large number of individual bytes with strides of 1, 64, and 4096. With a stride of 64, every access is to a new cache line. With a stride of 4096, every access is to a new virtual memory page. The results for the server used in our experiments are shown in Figure 1. The server has four Intel Xeon E7-4870 CPUs running at 2.40GHz (32nm Westmere), each with 10 cores. Each core has a 32KB L1 instruction cache, a 32KB L1 data cache and a 256KB L2 cache, while the cores on a chip share a 30MB L3 cache. The system uses rotating disk storage – 4 x 7200rpm drives in a RAID-10 configuration.

[1]Common in chips intended for server applications.
[2]https://software.intel.com/en-us/forums/intel-manycore-testing-lab/topic/287236 accessed 07 Jan 2016.
[3]www.ece.cmu.edu/calcm/isca2015

Table 1: Notation

p	bytes per payload
n	bytes per list next pointer
K	number of payloads in a list element
s	bytes for a particular list element
r	number of list elements in a run (with fixed K)
k	a parameter controlling settings of r and K
P	total number of postings for corpus
B	total number of bytes used for all linked lists
v	up to v postings are kept in the vocabulary entry

Figure 1: Times reported by the memory access benchmark for different block sizes and stride values of 1, 64 (next cache line) and 4096 (next VM page).



In turbo mode, two cores can run up to 2.80GHz. Since our benchmark is single threaded, the actual CPU speed should have been 2.80GHz. The server has 512GB of RAM and we experiment with allocated blocks of from 1GB to 32GB. As can be seen:

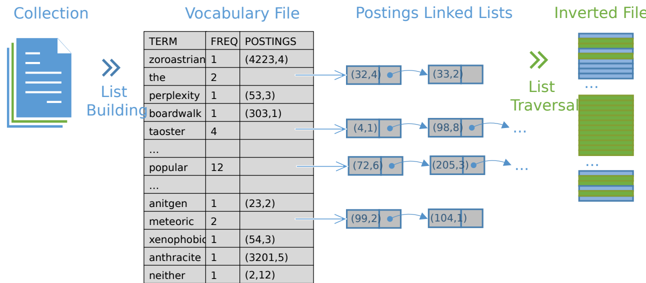
- The average access times for a stride of 1 are small (about 1.8 nanoseconds) and very consistent – maximum benefit is obtained from each 64-byte memory access.
- The access time for stride 4096 is considerably smaller for the 1GB block than for the larger block sizes. This is because the number of iterations in the benchmark means that in the stride of 4096 case the same locations are visited 64 times each. For the smaller strides there are no repeat visits. In the stride of 4096 case 262,144 locations are visited per GB. Thus all the visited locations can fit in L3 cache (30MB capacity, 491,520 cache lines) for the 1GB block size but not for any of the larger blocks.

We draw the conclusion from this that the primary goal of in-memory optimisation should be to minimise the number of 64-byte reads from RAM. Secondary goals of increasing larger scale locality of reference may reduce the cost of address translation, and increase the value of on-chip caches, but will not bring as much benefit.

1.3 Further details of indexing

During indexing a vocabulary structure such as a trie, b-tree, or hashtable is built up. In our case we use a power-of-two hash table whose capacity doubles if it becomes more than 90% full. We use the Fowler-Noll-Vo (FNV) hashing function [5] which is claimed to be fast, to have high dispersion and a low collision rate.

Figure 2: Indexing process: term postings ([document id, word position] tuples) are accumulated in the vocabulary hashtable and linked lists during List building and then stored in an inverted file during List traversal.



Each vocabulary item includes an occurrence count, plus pointers to the start and end of a linked list of postings. The end pointer supports efficient appending. In the simplest scheme (simplistic) each list element contains p bytes of payload (typically document number and term position within the document) plus n bytes to represent a pointer to the next element of the list. Memory for the list elements is obtained by allocating the next $s = p + n$ bytes from a large contiguous memory space. There is no space overhead in such allocations, and negligible time overhead.[4] If the total number of postings is P, then the total number of bytes allocated for linked lists in the simplistic scheme will be $B = P * s$. We focus on the performance of two phases only of the indexing process:

(1) List building: Appending postings to the linked list corresponding to term instances encountered during document scanning, and
(2) List traversal: Traversing the postings lists in order to write them in compressed form to the actual inverted file, once List building is complete. Note that a permutation array is used to allow the postings lists to be processed in alphabetic order of the terms to which they relate.

These phases are illustrated in Figure 2. Note that memory is continuously allocated during List building. Each newly created list element is contiguous with the one before it, except when switching between 64MB blocks. Memory demand grows monotonically and peaks at the end of this phase. All of the memory used during List building is accessed again during List traversal. List elements are accessed only once during List traversal. They will usually be accessed twice during List building, once at creation and once to adjust their NEXT pointer.

1.4 Memory access patterns

During the List building phase there are five main causes of memory accesses:

(1) Accessing instructions to perform steps (2) to (5) below;
(2) Scanning input text from a buffer set and extracting words;
(3) Looking up words in the vocabulary structure;
(4) Creating a new list element at the next available spot; and
(5) Accessing the previous tail of the list in order to insert a payload or to update its NEXT pointer.

Hopefully, the loop needed to carry out items 2–5 is tight enough to permit all instructions to fit into the instruction cache. Item 2 has good locality of reference. Item 3 is essentially random and the vocabulary structure is typically too large to fit into even the L3 cache. Successive creations of list elements at Item 4 have very good reference locality. Accessing the tail of lists at Item 5 has reasonably good locality on average because there is a good probability that the term whose latest occurrence is being recorded is one which was encountered recently enough that its previous posting was accessed recently. The major causes of memory accesses during List traversal are:

(1) Accessing instructions to perform steps (2) to (4) below;
(2) Accessing terms in the vocabulary structure;
(3) Following the chain of pointers from head to tail of the corresponding linked list.
(4) Writing postings into a fixed output buffer.

In this case Items 1 and 4 probably have good locality, Item 2 has similar locality to Item 3 for list building. However, Item 3 for List traversal has much worse locality of reference than Items 4 and 5 for List building because of large memory address gaps between successive elements of the same list. Consider, for example, an infrequently occurring term such as “zymurgy”. Between successive occurrences of “zymurgy” there may be a billion other term occurrences. In that case the memory address gap between successive elements of the simplistic linked list for “zymurgy” would be approximately s GB. We report times for single-threaded indexing to allow us to focus on the memory access properties of the linked list and to simplify analysis. Substantial reductions in overall elapsed time for indexing are to be expected when sharded sub-indexes are built in parallel and later merged.

2 OPTIMISATION TECHNIQUES.

We propose three different techniques for improving memory performance of the simplistic linked-list indexing method. We also mention a small trick which further improves performance.

Our starting point is the simplistic method in which a payload comprises a five-byte document number and a one-byte integer representing word position within the document. The use of only one byte is consistent with our focus on indexing short texts. Use of odd-numbers of bytes is already a significant space saving relative to using 64-bit and 32-bit integers. We have allowed seven bytes to represent the NEXT pointer, of which five-bytes is used as the actual pointer (sufficient to address one terabyte), since in compression methods described later in this section, the NEXT pointer is overloaded with book-keeping information. In other words, $p = 6, n = 7, s = p + n = 13$.

2.1 List chunking

Chunking is a means to reduce memory demand, and to improve locality of reference, by storing multiple payloads in a single list element. A chunked list element contains K (where $K > 1$) payloads but only one NEXT pointer. A counter, taking c bytes, may also be included to keep track of how many payloads have been used, but that space can be saved by computing the in-chunk count from

[4]Note: that some system implementations of memory allocation functions like malloc() impose a more than 100% space overhead for small requests. We avoid this by allocating memory in blocks of 64MB and by keeping our own pointer to the next available address.

Table 2: Details of the paper titles collection used in Table 3.

Number of documents	89.53 million
Number of terms	11.98 million
Number of term occurrences (postings)	1.004 billion

Table 3: Length expectancies: paper titles collection.

Length so far	Expected final length
1	83.7
10	844.6
100	4078.0
1000	18139.7
10000	82823.5
100000	461881.4
1000000	5004679.7
10000000	25372987.7

the overall count of postings stored, or by making use of the two bytes in the overloaded NEXT field. Overall there is a memory saving of $(K-1)*n$ bytes compared with an unchunked version, provided that the chunked element actually contains K payloads, i.e. it is fully used. All chunks but the last in each list will be fully used, but a substantial memory overhead may be incurred due to partially used chunks at the end of many postings lists. Choosing an optimal chunking scheme (Oracular) requires fore-knowledge of the distribution of postings list lengths, which is unrealistic in most settings.

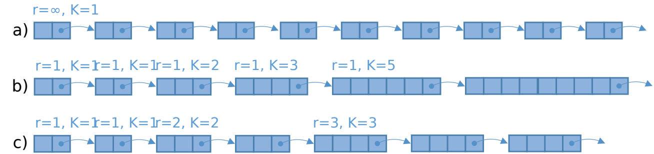
2.1.1 Distribution of list lengths. It is well known that the distribution of term occurrences in text is roughly Zipfian [15] – a few terms have very high frequency and very many occur very few times or even once only.

We computed a frequency histogram for the academic paper title collection described in Table 2, in which the independent variable is postings list length and the dependent variable is the frequency of occurrence of lists of that length. Sorting by descending length and accumulating both frequency and the product of length and frequency we can, for each length, compute the expected final length once a postings list has reached a particular length. Table 3 is like an abbreviated actuarial table for postings lists. It shows that the average length of postings lists in which we have already seen m postings increases sharply with increasing m. This suggests that a scheme in which the chunking factor K dynamically adapts to a growing list is likely to outperform one in which K is fixed. We explore this hypothesis experimentally later in the paper.

2.1.2 Methods considered. We consider chunking schemes that divide a postings list into runs of up to r chunks. All the chunks in a run have the same size (K). Both r and K may vary throughout a postings list. In the following we describe each of the chunking methods that we considered; Their characteristics are summarised in Table 5. Some methods are illustrated in Figure 3.

A scheme which implements a form of geometric growth (i.e. each K is a constant multiple of the previous one) performs quite well, but small tweaks can potentially improve on it. We experimented with $r > 1$ and with the Fibonacci series while Büttcher and Clarke [3] (See also Section 4.5 of Büttcher et al. [4] experiment

Figure 3: Selected chunking methods: a) Simplistic, b) FibonacciA (chunk size increases with the the Fibonacci series), and c) FibonacciB (in addition, run lengths increase with the Fibonacci series).



with limiting the maximum size of chunks, and with constant sizes for the first few postings. Note that the Fibonacci series approximates a geometric series in which the multiplier is approximately 1.618.

Simplistic – This is the unchunked method we want to improve upon. The number of list elements is equal to the number of postings. There will never be any space wasted in over-allocation but space for a pointer is required in every list element.

Oracular – If all terms and their frequency were known in advance, it would be possible to make “lists”, consisting of a single chunk containing all the payloads for a term, and with no space required for pointers. This is our "speed of light" baseline . For it to be practical an extra pass would be required through the text data, to calculate the term frequencies.

Fixed Blocks – Each list element contains a fixed number of payload, plus one pointer. This reduces the overhead due to pointers at the cost of over-allocation in the last list element.[5]

FibonacciA – As one can see from Table 3, the more postings already encountered for a particular term, the more additional postings can be expected to be accumulated in future. A strategy to match this expected growth pattern as closely as possible should use smaller chunk sizes when postings list lengths are small and final list lengths are uncertain, and larger chunk sizes when the list is already long, projecting an even larger amount of postings to be added in future. FibonacciA is an example of such a scheme; The number of postings per chunk grows according to the Fibonacci series. The run length is one, except for the first two elements.

FibonacciB – This method tracks the ideal curve (Oracular) more closely by not only increasing the chunk size in line with the Fibonacci Series, but also the length of runs with each chunk size.

Power based methods – The remaining methods outlined in Table 5 are based on exponential growth, either with a constant run length as in the methods Power1, Power5 and Power10, or with variable run lengths as in the methods Power, Subpower and Superpower.

Note also that we have also studied ways in which the parameters of the chunking model can be adapted to the overall characteristics of a data set as it is being indexed. Space does not permit us to discuss this. Figure 4 illustrates how some of the methods deviate from the Oracular allocation for postings lists of different lengths.

[5] An anonymous reviewer has suggested that the fixed chunk size should be aligned to cache size. This would be interesting to follow up in future work.

Figure 4: Ability of different chunking methods to match the minimal space requirements of a postings list.

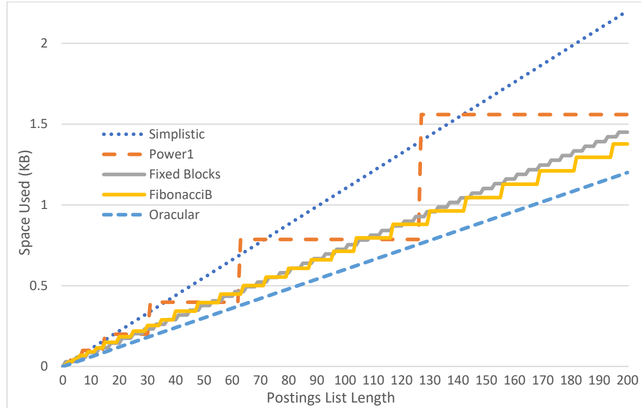


Table 4: PSG (and PPC) scores as the number of payloads stored in the vocabulary entry v ranges from 0 to 2, again for $p = 6$ and $n = 5$. Because $p > n$, storing two payloads in the vocabulary entry would require an increase in memory requirements for the vocabulary structure.

Scheme	$v = 0$	$v = 1$	$v = 2$
Oracular	1.833 (83.5)	1.855 (83.7)	1.864 (323.2)
FibonacciC	1.764 (34.0)	1.802 (50.4)	1.816 (58.7)

Table 5: Chunking methods explored on the academic paper titles collection, using $p = 6$ and $n = 5$. The subscript $_i$ refers to the ith run in a postings list. (Powers methods number from 0, while Fibonacci methods start at the 2nd element of the sequence: $1, 1, 2, 3, 5, \ldots$.) Where applicable, we explored k in the range $2 \ldots 8$; the best PSG scores are reported.

Method	r	K	PSG	PPC
Simplistic	∞	1	1.0	1.0
Fixed blocks(k=4)	∞	$2 * k$	1.549	7.5
FibonacciA	1	fib(i)	1.404	41.6
FibonacciB	fib(i)	fib(i)	1.763	**26.3**
FibonacciC	fib($i - 3$) or 1 if $i < 4$	fib(i)	1.764	34.7
Power(k=2)	k^i	k^i	1.764	28.6
SubPower(k=2)	k^{i-1}	k^i	1.764	33.3
SuperPower(k=2|3))	k^{i+1}	k^i	1.759	23.1
Power1(k=2)	1	k^i	1.278	44.2
Power5(k=2)	5	k^i	1.649	26.1
Power10(k=2)	10	k^i	1.701	20.6
Oracular			1.833	83.7

2.1.3 Comparison of chunking methods. We compare the chunking schemes using two measures (which may be in opposition to each other.)

- PSG: Potential Problem Size Gain relative to the Simplistic scheme (i.e. memory required by the Simplistic scheme divided by memory required by this one), and
- PPC: Average number of Postings Per Chunk.

A high number for the former indicates how much we could scale up the problem size for a given amount of RAM. A high value for the latter indicates a significant improvement in locality of reference during the List traversal phase. However, it may also have a deleterious effect during List building since the next posting to be written may be an address within memory allocated long ago.

Table 5 reports the PSG and PPC values obtained for each method, based on $p = 6, n = 5$ (payload size and pointer size, respectively). In our experiments, we actually used $p = 6, n = 7$. Those values do not change the PPC values in the table, but they do increase the PSG values. For example, the PSG values for Oracular, and FibonacciB increase to 2.167 and 2.058 respectively. For all of the Power methods and the fixed block method we explored values of $2 <= k <= 8$ and reported the results for the value of k which achieved the highest PSG score. From Table 5 it is clear that almost any form of chunking produces a worthwhile increase in PSG and PPC.

The dynamic families (Power and Fibonacci) of methods perform worse when runs are too short. This is illustrated in the results for Power1 where PPC achieves very high values, but PSG starts low and decreases rapidly as K increases. Essentially we are trying to choose values of K which are consistent with the length-expectancy data, illustrated in Table 5. At a particular point in indexing, the best choice for the next value of K should relate to the distribution of extra length yet to come.

The equal best PSG score was achieved by FibonacciC (starting with the values 1 and 2) and Power (for $k = 2$) but the PPC value of FibonacciC was higher. Accordingly the FibonacciC method of chunk allocation seems most promising, but we note that several other dynamic methods achieve very similar performance, and may prove to be superior on other collections.

2.2 In-memory compression

A useful technique for compressing postings lists converts a sequence of document numbers into a sequence of document number differences and represents the differences using a variable length code. There is a substantial literature on compression techniques for postings lists. Trotman [13] identifies three generations: byte-encoding, word-aligned codes, and SIMD codecs, and empirically compares several representative examples. We elected to use the very straightforward variable byte (v-byte) method due to Scholer et al. [11] for its simplicity, and ease of coding. As noted in Section 2, each posting uses a single byte to represent the word position within a document – we assume short documents such as academic paper titles. This makes our work incomparable with studies of compression effectiveness, and we do not report bits per posting results. Once in-memory inversion is complete, the final index can be compressed, in any way desired, to minimise space, or to maximise decompression speed.

A problem with using compression schemes in conjunction with chunking is that at the time of allocating memory for the next chunk, the number of bytes needed is unknown because it depends upon the gaps between term occurrences. There is also the issue that use of v-bytes may result in wasted bytes at the end of a chunk, because the number of bytes left in the chunk may be non-zero, but less than the number of bytes needed for the next posting. It would

be possible to allow the bytes of a compressed posting to extend to the next chunk, but this would complicate and likely slow down the code to store and decompress. We decided to accept a small amount of wasted space due to this cause.

2.3 Using "Large Pages"

As noted in Section 1.1, the performance of a process running on a virtual memory operating system such as Windows, Linux or Solaris may be significantly reduced by TLB misses. Karakostas et al. [10] report that TLB misses can add 50% to the execution time of large-memory programs. Those authors describe an effective and flexible method for reducing TLB misses but unfortunately their improvement is not available commercially. What is available in the type of hardware environment we envisage is a more rigid system called *large pages* or *huge pages* depending upon the operating system. In this system, the page size is increased by a factor of 512, from 4kB to 2MB. This effectively eliminates the need for the lowest level of the page table and means that a single entry in the TLB can replace the 512 entries which would otherwise be required to record the same segment of the mapping.

On the negative side, the physical memory to satisfy a request for a large page must be contiguous. This can cause allocation failures in circumstances where sufficient pages are actually available. Note that Gog and Petri [6] report further benefits from using 1GB "huge pages" but these were not available in our environment. We expect similar benefit from the Karakostas et al. approach, but with less risk of allocation failures.

2.4 Another trick: Payloads in vocabulary entries

We noted above that the vocabulary entries contain two pointers to list elements. The space for the two pointers can potentially be used to store v payloads when postings lists are very short (see Trotman et al. [14]). I.e. if the term frequency were one, then no list would be needed and the payload could be stored in place of the list pointers. The first v postings for each term are stored in the vocabulary entry. When the $(v + 1)$-th occurrence is found, a list is set up and payloads are copied into it. We have implemented this for $v = 1$ and find that it brings a small degree of benefit.

2.5 Discussion of chunking

The advantages of the best chunking method over SIMPLISTIC promise to be quite impressive. While keeping single postings in the vocabulary hashtable we observe for the best method:

(1) A gain of 80% in the size of problem which can be handled in a given amount of RAM;
(2) A reduction by a factor of approximately 50 in the number of pointers which need to be followed during LIST TRAVERSAL.

Table 4 shows that the FIBONACCIC scheme approaches the ORACULAR PSG scores quite closely. The performance figures above were all obtained from the term frequency distribution for a particular collection (Academic) of relatively modest size ($P = 1.004 * 10^9$).

3 EXPERIMENTAL METHODOLOGY

3.1 Hardware and Software

We study an indexer written in C which implements SIMPLISTIC, FIXED BLOCKS, SUB-POWER and FIBONACCIB methods. It provides options to turn on and off:

- Storing up to 2 postings in vocabulary entries.
- v-byte compression within chunks.
- The use of large pages for hashtable and list storage.
- The writing of indexing output files (to allow timing of the parts of the algorithm we wanted to study.)

We compiled the code on Visual Studio 2013 with full optimisation and ran it on the server described in Section 1.2, running Windows Server 2012R2. Our indexing runs were single-threaded.

3.2 Time and space measurements

We separately measured the elapsed (wall-clock) times for both LIST BUILDING and LIST TRAVERSAL phases. We reported the amount of memory allocated to the hash table and to the postings lists (linked lists, postings array, or dynamic arrays.). We ran each condition one after the other on an otherwise idle machine[6] and recorded elapsed times for the two phases. We repeated the sequence of runs three times, and report the median of the observed times.

To reduce I/O times:

(1) We memory mapped the file of documents to be indexed and pre-scanned the file prior to each indexing run to ensure its pages were in memory.
(2) During the LIST TRAVERSAL phase, i/o to the vocabulary file and inverted file was disabled.

3.3 Data sets

We study the following collections of (natural language) text titles:

Wikipedia: 11 million page titles from the English Wikipedia.
Academic: 89 million titles of academic papers. This was the collection whose term frequency list was used to compare chunking methods.
Webpages: 1.3 billion titles of web pages.
Samples: Randomly chosen samples of the Webpages dataset in the following proportions: $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{64}$, $\frac{1}{128}$.

We also study two artificial collections engineered to have the same problem size as Webpages but very different term frequency distributions. This allows us to investigate whether findings on real data sets are also true for different term frequency distributions.

Uniform A sequence of 46,682 distinct terms is cycled through over and over to create a collection of 1.004 billion postings. This creates a uniform distribution of term frequencies, and constant gaps between postings for the same term. E.g. "AAA AAB AAC AAD...". **Uniform50B** is a much larger version expanded to 50 billion postings.
Skewed Created in the same way as Uniform, but a single term 'Q' is inserted before every term occurrence. E.g. "Q AAA Q AAB Q AAC Q AAD...". The total number of postings is identical to Uniform and the term frequency distribution

[6]Note that no special steps were taken to ensure this and that all normal Windows services were running.

Table 6: Peak memory usage for linked lists (in MB). PSG values for List building phase are in parentheses, compared to Simplistic run (parsimonious payloads and pointers).

	Wikipedia	Academic	Web pages
Problem size	3.27×10^7	1.004×10^9	8.46×10^9
Simplistic (p=12,n=8)	623.8	19,141.3	161,275.1
Simplistic	343.1	10,527.7	88,701.3
FibonacciB	221.2 (1.55)	5971.8 (1.76)	49,619.2 (1.79)

is uniform except for the extreme outlier 'Q' which has a frequency of more than 500 million.

3.4 Results: Increasing problem size

Tables 6 and 7 compare Simplistic and FibonacciB chunking methods both with and without use of Large Pages (1MB) on memory space and elapsed times for both indexing phases for a range of problem sizes. The largest of the three collections is 259 times larger than the smallest. We make the following observations:

(1) Memory space savings due to FibonacciB chunking are approximately what we expected for Academic (1.76 v. 1.73). We are not sure why there is this small difference.
(2) Memory space savings are very close to this value for the largest collection (Web pages). They are quite a bit smaller for Wikipedia but still substantial.
(3) Line 1 of Table 6 shows that a simple-minded implementation of lists (using 64-bit integers for pointers and document numbers, and 32-bit integer for word positions) without chunking would require 157.5GB of memory for linked lists for the Web pages corpus instead of the 48.5GB in Line 3.
(4) We expected that the use of chunking might slow down the List building phase. However, observed differences are small and in the case of Web pages, the List building time actually reduced by 2.5 percent.
(5) The speed-ups due to chunking for List traversal were quite dramatic – up to a factor of 5.65 for the largest collection. Relatively little benefit was seen on the smallest collection (speed-up by a factor of 1.95) perhaps because the memory requirements were not large enough to trigger significant page table access effects.
(6) The use of Large Pages was beneficial in both phases with the one exception of List building for the Web pages collection. However there was quite a large spread of times for this collection, so this observation may be due to noise. In List traversal the gains from large pages were much smaller than those from use of FibonacciB chunking.

3.5 Results: Varying term freq. distribution

Table 8 show that memory gains and speed-ups during List traversal due to chunking are relatively consistent for problems of the same size regardless of the distribution of term frequencies. This reassures us that strong gains are achieved from FibonacciB regardless of term frequency distribution, despite the fact that the method was designed to match the characteristics of a Zipfian distribution. Of course, if we knew that the term frequency distribution was uniform, much bigger gains would be possible through the use of large fixed size chunks. The fifth column reports an experiment in which the Uniform dataset was scaled up by a factor of 50 to 50 billion postings. No results are shown for the simplistic method because memory allocation failed. Time to build the lists scaled almost linearly. The ratio of list building times was $\frac{29667.0}{554.9} = 53.5$ rather than the expected 50.0. Time to scan the lists did not scale, growing by a factor of $\frac{3081.7}{22.0} = 140.0$. However, when the indexer read the input file one megabyte at a time into a ring of buffers, rather than mmap-ing the whole file, list scanning time dropped to 981.9 seconds, a ratio of 44.6.

3.6 Scalability

Table 9 shows how memory usage and elapsed times grow with the size of the collection. Growth in memory requirements is slightly sub-linear, presumably because the FibonacciB scheme requires fewer bytes per posting in longer lists than shorter ones and there are more long lists in a larger collection. Growth in List traversal times is markedly sublinear possibly due to the same effect.

The most notable feature of the table is the row of build times. While the build time increases less than expected when moving from Wikipedia to Academic, from Academic to Web titles it grows by a factor of 31 while the problem size grows only by a factor of 8.4. Our suspicion is that this may be due to inefficiencies in memory mapping the 130GB input file. Clearly the cause of this blow-out in time needs to be discovered and rectified, but this is left for future work.

4 DISCUSSION AND CONCLUSIONS

Memory reductions and indexing speed-ups are what we actually want to optimise. We have focused on relatively high-level performance measures, such as elapsed time and memory usage. Further gains would likely flow from close scrutiny of instructions per clock, and rates of cache misses, branch mispredictions, and TLB misses. In this work, we have consciously ignored the efficiency of the vocabulary structure and its memory-access interaction with the performance of the linked lists. This aspect would be worth exploring further in future work.

In the situation where unchunked lists fit into available RAM, chunking speeds up List traversal by a solid factor due to in-memory compactness and locality of reference. In all cases where the unchunked lists are too big to fit into available RAM, the advantage of chunking is obviously much greater. Even when a very large data structure is entirely resident in memory, the need to map virtual memory addresses to physical ones, whenever the sequences of accesses moves from one page to another, can cause significant slow downs. The same guidelines apply to in-memory structures as to ones on disk:

- Keep structures compact;
- Try to achieve maximum locality of reference.

These guidelines also serve to magnify the value of CPU caches and of memory systems which read a cache line (usually 64 bytes) at a time. We have shown that the use of compact representations of payloads and pointers combined with a dynamic chunking scheme dramatically reduces memory requirements. It also dramatically reduces the time taken to scan the linked lists without any harm to the time to build them.

Table 7: List building and List traversal elapsed times (in seconds) for the three natural collections. Neither chunking nor use of large pages has much effect on the List building times. The slow times for the Web titles collection is to unexpectedly slow I/O times for reading the text to be indexed. Speed-up ratios relative to Simplistic are shown in parentheses.

Measurement of	Method	Wikipedia	Academic	Web titles
Problem size		3.27×10^7	1.004×10^9	8.46×10^9
List building	Simplistic	16.0	335	10,487
	Simplistic (Large Pages)	14.7	312	10,740
	FibonacciB	15.8	337	10,472
	FibonacciB (Large Pages)	14.7	317	10,333
List traversal	Simplistic	9.60	211.0	2171
	Simplistic (Large Pages)	8.70 (1.10)	166.5 (1.27)	1529 (1.42)
	FibonacciB	5.20 (1.85)	48.8 (4.32)	426 (5.10)
	FibonacciB (Large Pages)	**5.00** (1.92)	**42.6** (4.95)	**384** (5.65)

Table 8: Peak memory for linked lists as well as List building and List traversal elapsed times for the Academic and three artificial collections. PSG memory use values for List building phase only are in parentheses.

Measurement of	Method	Academic	Uniform	Skewed	Uniform50B
Problem size		1.004×10^9	1.004×10^9	1.004×10^9	5×10^{10}
Peak memory (MB)	Simplistic	10527.7	10527.7	10527.7	–
	FibonacciB	5971.8 (1.76)	5831.2 (1.81)	5792.5 (1.82)	286656.0
List building (seconds)	Simplistic	335.3	560.4	336.6	–
	FibonacciB	337.1	554.9	340.3	29667.0
List traversal (seconds)	Simplistic	211.0	88.1	64.1	–
	FibonacciB	48.8	22.0	18.4	3081.7

Table 9: Scaling with problem size using FibonacciB chunking without Large Pages. Memory sizes are the peak memory use for linked list elements.

	Wikipedia	Academic	Web pages
Problem size	3.27×10^7	1.004×10^9	8.46×10^9
Problem size	1.0	30.7	258.7
Memory size	1.0	27.0	224.3
Build time	1.0	21.3	662.8
Scan time	1.0	9.38	81.9

Large pages bring modest benefits which add to those brought by other techniques. However, operating system support for large virtual memory pages comes with a number of restrictions.

Somewhat to our surprise, use of v-byte compression within chunks made relatively little difference to the speed of either phase. However, the further halving of the memory required to store linked lists argues in it is favour.

The best combination we found employs a combination of storing very short postings lists in vocabulary entries, Fibonacci dynamic chunking, compression within chunks, and large VM pages.

REFERENCES

[1] AnandTech. Anandtech website. www.anandtech.com, Accessed 2017.
[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books, 0.90 edition, March 2015.
[3] S. Büttcher and C. L. Clarke. Memory management strategies for single-pass index construction in text retrieval systems. *University of Waterloo Technical Report CS-2005-32*, 2005.
[4] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and evaluating search engines.* MIT Press, Cambridge, MA, 2010.
[5] G. Fowler, P. Vo, and L. C. Noll. FNV hash. Website, Accessed 13 July 2015. www.isthe.com/chongo/tech/comp/fnv/. Accessed 13 July 2015.
[6] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.
[7] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. BitFunnel: Revisiting signatures for search. In *Proceedings of SIGIR 2017*, pages 605–614, New York, NY, USA, 2017. ACM.
[8] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *JASIS*, 41:581–589, 1990. https://www.asis.org/Publications/JASIS/Best_Jasist/1991CandelaandHarman.pdf.
[9] Intel. Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm, Accessed 2017.
[10] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. Redundant memory mappings for fast access to large memories. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015.
[11] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of SIGIR 2002*, SIGIR '02, pages 222–229, New York, NY, USA, 2002. ACM.
[12] M. Taube. *Studies in coordinate indexing.* Documentation Incorporated, 1953.
[13] A. Trotman. Compression, SIMD, and postings lists. In *Proceedings of the 2014 Australasian Document Computing Symposium*, page 50. ACM, 2014.
[14] A. Trotman, X. Jia, and M. Crane. Managing short postings lists. In *ADCS*, pages 113–116, 2013.
[15] G. K. Zipf. *Human behavior and the principle of least effort.* Addison-Wesley, 1949.
[16] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.