An Exploration of Postings List Contiguity in Main-Memory Incremental Indexing

Nima Asadi[1,2], Jimmy Lin[3,2,1]

[1]Dept. of Computer Science, [2]Institute for Advanced Computer Studies, [3]The iSchool
University of Maryland, College Park

ABSTRACT

For text retrieval systems, the assumption that all data structures reside in main memory is increasingly common. In this context, we present a novel incremental inverted indexing algorithm for web-scale collections that directly constructs compressed postings lists in memory. Designing efficient in-memory algorithms requires understanding modern processor architectures: in this paper, we explore the issue of postings list contiguity. Postings lists that occupy contiguous memory regions are preferred for retrieval, but maintaining contiguity is costly in terms of speed and complexity. On the other hand, allowing discontiguous index segments simplifies index construction but decreases retrieval performance. Understanding this tradeoff is our main contribution: We show that co-locating small groups of inverted list segments yields query evaluation performance that is statistically indistinguishable from fully-contiguous postings lists. In other words, we can achieve ideal performance with a relatively small amount of effort.

1. INTRODUCTION

For text retrieval applications today, it is reasonable to assume that all index structures fit in main memory. In this context, we explore incremental (sometimes referred to as "online") inverted indexing algorithms in main memory for modern web-scale collections. We describe a novel indexing algorithm for incrementally building compressed postings lists directly in memory. Of course, incremental indexing is not a new research topic, but most previous work assumes that the index will not fit in memory and must reside on disk. Our assumption puts us in a different, underexplored region of the design space.

Frequently, indexing algorithms encode a tradeoff between indexing and retrieval performance. Our study specifically examines the issue of postings list contiguity, which manifests such a tradeoff. By contiguity we mean whether each postings list occupies a single block of memory or is split into multiple segment placed at different memory locations. Why does contiguity matter? From the retrieval perspective, we would expect an impact on query evaluation speed: traversing postings lists that occupy a contiguous block of memory takes advantage of cache locality and processor prefetching, whereas discontiguous postings lists suffer from cache misses due to pointer chasing. However, from the indexing perspective, maintaining contiguous postings lists introduces substantial complexity, for example, requiring either a two-pass approach, eager over-allocation of memory, or repeatedly copying postings when they grow beyond a certain size. With each of these techniques we would expect indexing performance to suffer. Thus, a faster, simpler indexing algorithm that does not attempt to maintain postings list contiguity may result in slower query evaluation. It is this tradeoff that we seek to understand in more detail.

LSDS-IR'14, February 28, New York NY, USA.

This paper has two main contribution: First, we present a novel in-memory incremental indexing algorithm with several desirable features: it is fast, scales to modern web-scale collections, and takes advantage of best practice index compression techniques. Second, in the context of this indexing algorithm, we explore the impact of postings list contiguity on indexing and query evaluation performance (both conjunctive and disjunctive). We find that small discontiguous inverted list segments do indeed cause a drop in query evaluation speed, but that co-locating small groups of index segments yields performance that is statistically indistinguishable from fully-contiguous postings lists. In other words, we can achieve ideal performance with a relatively small amount of effort.

2. BACKGROUND AND RELATED WORK

Traditional disk-based indexing algorithms are tuned to the performance characteristics of rotational magnetic disks. Similarly, efficient in-memory indexing algorithms must take into account the design of modern processor and memory architectures. The biggest challenge we face is the so-called "memory wall" [2]: for several decades now, increases in processor speed have far outpaced improvements in memory latency. This means that RAM is becoming slower relative to the CPU. In the 1980s, memory latencies were on the order of a few clock cycles; today, it could be several hundred clock cycles. In modern architectures, memory latencies are hidden by hierarchical caches, but cache misses remain expensive. Thus, it is imperative that developers either carefully manage data locality or structure memory access patterns to take advantage of prefetching. This suggests that discontiguous postings lists will slow query evaluation considerably due to pointer chasing. However, there is one nuance worth noting: Following best practice, we use PForDelta [26, 24] for compressing postings lists. Since it is a block-based technique, decompression yields memory access patterns that differ from techniques which code one integer at a time (e.g., variable-length integers, γ codes, etc.). Our experiments show that this has the effect of partially masking memory latencies.

Inverted indexing for text retrieval has been thoroughly studied by researchers, and a comprehensive review is beyond the scope of this work; we refer the reader to a survey for more details [25]. Most previous work on *incremental* indexing assumes that postings lists do not fit in memory and ultimately must be organized on disk. Tomasic et al. [21] was among the first to explore the design space of indexing strategies in this scenario: options they explored included writing discontiguous index segments to disk and different ways of "reserving" extra space on disk to accommodate future postings list growth. Elaborations and variations of the same basic ideas have been explored by others [4, 17, 8, 18].

We compare our indexing approach to that of Lester et al. [12, 13], which begins with in-memory inversion within a buffer [11]. The authors present three options for what to do once memory is exhausted: rebuild the index on disk from scratch (not very practical), modify postings in place on disk (practical only for small updates), or to selectively merge in-memory and on-disk segments and rewrite to another region on disk. In particular, they explored a geometric partitioning and hierarchical merging strategy that limits the number of outstanding partitions, thereby controlling query costs. The same basic idea was described at around the same time by Büttcher et al. [6], who called their approach "logarithmic merge". Both approaches were subsequently generalized and extended [10, 15]

Our work is most similar to Fontoura et al. [9], who examined term-at-a-time and document-at-a-time approaches for query evaluation using in-memory indexes. Their work, however, assumes contiguous postings (with an unspecified compression technique) and focuses on query optimization, whereas we adopt baseline query evaluation algorithms but explore the memory fragmentation and indexing speed issues. Prior to that, Strohman and Croft [19] also studied query evaluation strategies in main memory. Another point of comparison is Earlybird [5], Twitter's in-memory real-time search engine. The system takes advantage of tweet-specific features and adopts a federated architecture consisting of small in-memory index segments—representing a different point in the design space.

3. APPROACH

3.1 Basic Incremental Indexing Algorithm

Our indexer consists of three main components, depicted in Figure 1: the dictionary, buffer maps, and the segment pool. The basic indexing approach is to accumulate postings in the buffer maps in an uncompressed form until the buffers fill up, and then to "flush" the contents to the segment pool, where the final compressed postings lists reside. Note that in this approach the inverted lists are discontiguous; we return to address this issue in Section 3.2.

The dictionary is implemented as a hash table with a bitwise hash function [16] and the move-to-front technique [23], mapping terms (strings) to integers term ids. The dictionary additionally holds the document frequency (df) for each term, as well as a head and tail pointer into the segment pool (more details below). In our implementation, term ids are assigned sequentially as we encounter new terms.

A *buffer map* is a one-to-one mapping from term ids to arrays of integers (the buffers). Since term ids increase monotonically, a buffer map can be implemented as an array of pointers, where each index position corresponds to a term id,

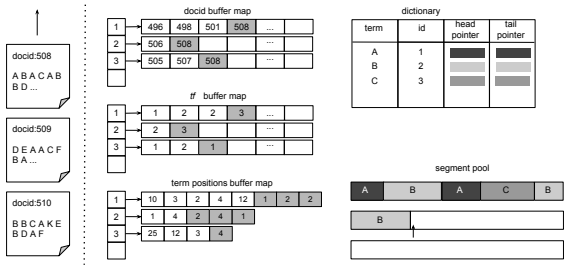

Figure 1: A snapshot of our indexing algorithm. In the middle we have buffer maps for storing docids, *tf*s, and term positions: the gray areas show elements inserted for document 508. Once the buffer for a term fills up, an inverted list segment is assembled and added to the end of the segment pool and linked to the previous segment via addressing pointers. The dictionary maps from terms to term ids and holds pointers to the heads and tails of the inverted list segments in the segment pool.

and the pointer points to the associated buffer. The array of pointers is dynamically expanded to accommodate more terms as needed. To construct a positional index, we build three buffer maps: the document id (docid) map, the term frequency (tf) map, and the term positions map. As the names suggest, the docid map accumulates the document ids of arriving documents, the tf map holds term frequencies, and the term positions map holds term positions. There is a one-to-one correspondence between entries in the docid map and entries in the tf map (for each term that occurs in a document, there is exactly one term frequency), but a one-to-many correspondence between entries in the docid map and entries in the term positions map (there are as many term positions in each document as the term frequency).

In the indexing loop, the algorithm receives an input document (sequentially numbered), parses it to gather all term frequencies and term positions (relative to the current document, starting from one) for all unique terms, and then iterates over these terms, inserting the relevant information into each buffer map. Whenever we encounter a new term, the algorithm initializes an empty buffer in each buffer map for the corresponding term id. Initially, the buffer size is set to the block size b that will eventually be used to compressed the data (leaving aside an optimization we introduce below to control the vocabulary size). Following best practices today, we use PForDelta [26, 24], with the recommended block size of $b = 128$. The term positions map expands one block at a time when it fills up to accommodate more positions. When the docid buffer for a term fills up, we "flush" all buffers associated with the term, compressing the docids, term frequencies, and term positions into what we call an inverted list segment, described below:

Each inverted list segment begins with a run of docids, gap-compressed using PForDelta; call this D. By design, the docids occupy exactly one PForDelta block. Next, we compress the term frequencies using PFor; call this F. Note that term frequencies cannot be gap-compressed, so they are left unmodified. Finally, we process the term positions, which are also gap-encoded, relative to the first term position in each document. For example, if in d_1 the term was found at positions $(1, 5, 9)$ and in d_2 the term was found at positions $(3, 16)$, we would code $(1, 4, 4, 3, 13)$. The term positions can

be unambiguously reconstructed from the term frequencies, which provide offsets into the array of term positions. Since the term positions array is likely longer than b, the compression block size, the term positions occupy multiple blocks. Call the blocks of term positions $P_1 \ldots P_m$.

Finally, all the data are packed together in a contiguous block of memory as follows:

$$[\,|D|,\ D,\ |F|,\ F,\ \{|P_i|,\ P_i\}_{0 \le i < m}]$$

where the $|\cdot|$ operator returns the length of its argument. Since all the data are tightly packed in an otherwise undelimited array, we need to explicitly store the lengths of each block to properly decode the data during retrieval.

Each inverted list segment is written at the end of the segment pool, which is where the compressed inverted index ultimately resides. Conceptually, the segment pool is an bounded array with a pointer that keeps track of the current "end", but in practice the pool is allocated in large blocks and dynamically expanded as necessary. In order to traverse a term's postings during query evaluation, we need to "link" together the discontiguous segments. The first time we write a segment for a term id, we add its address (byte offset in the segment pool) to the dictionary, which serves as the "head" pointer (the entry point to postings traversal). In addition, we prepend to each segment the address (byte offset position in the segment pool) of the next segment in the chain. This means that every time we insert a new segment for a term, we have to go back and correct the "next pointer" for the last segment. We leave the next pointer blank for a newly-inserted segment to mark the end of the postings list for a term; this location is stored in the "tail pointer" in the dictionary. Once the indexer has processed all documents, the remaining contents of the buffer maps are flushed to the segment pool in the same manner. By default, we build full positional indexes, but our implementation has an option to disable the term position buffers if desired. In this case, the inverted list segments will be smaller, but other aspects of the algorithm remain exactly the same.

Note that we currently do not handle document updates or deletes, but we envision relatively straightforward strategies that could be implemented in the future. It is possible to store document deletes in a separate data structure and ensure that deleted documents are not returned at query time (this is sometimes called the "tombstone" approach). Updates can be treated as deletes followed by additions. Periodically, the postings list can be rewritten with the deletes "merged" back in—this might happen when the indexes are persisted to durable storage for fault tolerance.

What are the implications of our index design? On the positive side, all data in the segment pool are "tightly packed" for maximum efficiency in memory utilization. During indexing we guarantee that there is no heap fragmentation, which may be a possibility if we simply used `malloc` to allocate space for each inverted list segment. On the negative side, postings traversal becomes an exercise in pointer chasing across the heap, without any predictable access patterns that will aid in processor prefetching across segment boundaries. Thus, as a query evaluation algorithm consumes postings, it is likely to encounter a cache miss whenever it reaches the end of a segment, since it has to follow a pointer.

One final optimization detail: we control the size of the term space by discarding terms that occur fewer than ten times (an adjustable document frequency threshold). This is accomplished as follows: instead of creating a buffer of length b when we first encounter a new term, we first allocate a small buffer equal to the *df* threshold. We buffer postings for new terms until the threshold is reached, after which we know that the term will make it into the final dictionary, and so we reallocate a buffer of length b. This two-step process reduces memory usage substantially since there are many rare terms in web collections.

3.2 Segment Contiguity

It is clear that our baseline indexing algorithm generates discontiguous inverted list segments. In order to create contiguous inverted lists, we would need an algorithm to rearrange the segments once they are written to the segment pool. Before going down this path, however, we first examined the extent to which contiguous segments would improve retrieval efficiency, from better reference locality, prefetch cues provided to the processor, etc. Let us assume we have an oracle that tells us exactly how long each inverted list is going to be, so that we can lay out the segments end-to-end, without any wasted memory. We simulate this oracle condition by building the inverted index as normal, and then performing in-memory post-processing to lay out all the inverted lists contiguously. The oracle allows us to establish an upper bound on query evaluation speed.

We propose a simple yet effective approach to achieving increasingly better approximations of contiguous postings lists. Instead of moving compressed segments around after they have been added to the segment pool, we change the memory allocation policy for the buffer maps. Whenever the docid buffer for a term becomes full (and thus compressed and flushed to the segment pool), we expand that term's docid and *tf* buffers by a factor of two (still allowing the term positions buffer to grow as long as necessary). This means that after the first segment of a term is flushed, new docid and *tf* buffers of length $2b$ replace the old ones; after the second flush, the buffer size increases to $4b$, and then $8b$, and so on. When a buffer of size $2^m b$ becomes full, the buffer is broken down into 2^m segments, each segment is compressed as described earlier, and all 2^m segments are written at the end of the segment pool contiguously—this strategy allows long postings to become increasingly contiguous, without wasting space on large buffers for rare terms.

To prevent buffers from growing indefinitely, we set a cap on the length of docid and *tf* buffers. That is, if the cap is set to $2^m b$, then when the buffer size for a term reaches that limit, it is no longer expanded. This means that the maximum number of contiguous segments allowed in the segment pool is 2^m. We experimentally examine the effect of m on query evaluation speed.

4. EXPERIMENTAL SETUP

Our experiments used the first English segment of the ClueWeb09 web crawl by Carnegie Mellon University (∼50 million documents, 247GB compressed). For evaluation, we used two sets of queries: the TREC 2005 terabyte track "efficiency" queries, which consist of 50,000 queries total, and a set of 100,000 queries sampled randomly from the AOL query log. Experiments were performed on a server running Red Hat Linux, with dual Intel Xeon "Westmere" quad-core processors (E5620 2.4GHz) and 128GB RAM.

Our indexer, called Zambezi, is implemented in C, compiled using `gcc` with `-O3` optimizations; it is currently single-

Query	1b	2b	4b	8b	16b	32b	64b	128b	Contiguous
Terabyte	49.7 (±0.2)	47.1 (±0.1)	45.9 (±0.4)	44.4 (±0.5)	42.9 (±0.4)	42.0 (±0.3)	41.6 (±0.1)	41.6 (±0.4)	41.3 (±0.1)
AOL	87.5 (±1.6)	83.2 (±0.5)	80.7 (±0.3)	75.5 (±0.5)	75.7 (±0.8)	75.8 (±0.3)	75.2 (±0.2)	75.0 (±0.6)	75.3 (±1.2)

Table 1: Average query latency (in milliseconds) for postings intersection using SvS with different buffer length settings. Results are averaged across 5 trials, reported with 95% confidence intervals.

threaded. To support the reproducibility of experiments reported in this paper, the system is released under an open-source license.[1] Since we focus on indexing, we wished to separate document parsing from index construction. Therefore, we assumed that documents have already been parsed, stemmed, with stopwords removed. Our reports of indexing speed do not include document pre-processing time.

We examined three aspects of performance: memory usage, indexing speed, and query evaluation latency. The first two are straightforward, but we elaborate on the third. For each indexer configuration, we measured query evaluation speed in terms of query latency for two retrieval strategies: conjunctive retrieval using the SvS algorithm, demonstrated by Culpepper and Moffat [7] to be the best approach to postings intersection, and disjunctive query processing using the WAND algorithm [3], which represents a strong baseline for top k retrieval (with BM25). Our implementations follow the two cited references, except for two substantive differences: the search procedures for finding docids were adapted to our discontiguous postings, and to probe a particular posting, the entire block is decompressed. Note that we did not implement the query evaluation optimizations of Fontoura et al. [9] and our indexes contained no auxiliary structures such as skip-lists. For both conjunctive and disjunctive retrieval we first indexed the entire collection, and then performed query evaluation at the end.

Since our focus is not on query evaluation, we believe that experiments with SvS and WAND are sufficient to illustrate the tradeoffs of our indexing algorithm. We do not consider any learning to rank [14] because it represents an orthogonal issue. In a modern multi-stage web search architecture [1, 22], an initial retrieval stage (e.g., using SvS or WAND) generates candidate documents that are then reranked by a machine-learned ranking model.

We compared our indexer against two open-source engines: Zettair (v0.9.3), which implements the geometric partitioning approach [13] and Indri (v5.1) [20]. To ensure a fair comparison, we disabled document parsing and used the already-parsed documents as input. As with our system, reports of indexing speed do not include time spent on document pre-processing.

5. RESULTS

5.1 Query Latency

Table 1 summarizes query latency for conjunctive query processing (postings intersection with SvS). The average latency per query is reported in milliseconds across five trials along with 95% confidence intervals. Each column shows a different indexing condition: 1b is the baseline algorithm presented in Section 3.1 (discontiguous postings). Each of $\{2, 4, 8 \ldots 128\}b$ represents a different upper bound in the buffer map growing strategy described in Section 3.2. The final column marked "contiguous" denotes the oracle condition in which all postings are contiguous.

[1] `http://zambezi.cc/`

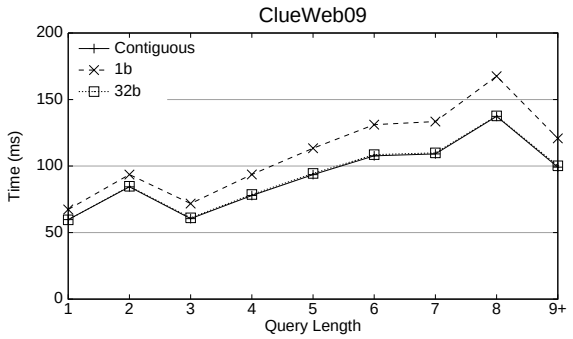

Figure 2: Query latency using SvS on AOL queries, by query length for different buffer length settings.

Query	1b	32b	Contiguous
Terabyte	150.0 (±0.5)	141.1 (±0.6)	141.1 (±0.2)
AOL	455.7 (±5.1)	434.3 (±5.8)	432.6 (±4.9)

Table 2: Average query latency (in milliseconds) to retrieve the top 1000 hits in terms of BM25 using WAND (5 trials, with 95% confidence intervals).

From these results, we see that, as expected, discontiguous postings lists (1b) yield slower query evaluation. As b increases, we allow the buffer maps to increase in length, lowering query latencies. At 32b, query evaluation performance is statistically indistinguishable from the performance upper bound (i.e., confidence intervals overlap). That is, we only need to arrange inverted list segments in relatively small groups of 32 to achieve ideal performance.

Figure 2 illustrates query latency by query length for the AOL query set under different conditions. Not surprising, the latency gap between contiguous and the 1b condition widens for longer queries. On the other hand, the difference between a contiguous index and the 32b condition is indistinguishable across all query lengths—the lines practically overlap in the figure.

For disjunctive query processing, we used WAND to retrieve the top 1000 documents using BM25. Table 2 summarizes these results. For space considerations, we only report results for select buffer length configurations. These numbers are consistent with the conjunctive processing case. A maximum buffer size of 32b yields query latencies that are statistically indistinguishable from a contiguous index. Note that the performance difference between contiguous postings lists and 1b discontiguous postings lists is less than 7%. In other words, there is much less performance degradation than in the SvS case.

The major finding of these experiments is that a relatively modest amount of postings contiguity suffices to yield performance that is indistinguishable from contiguous postings. From the processor architecture perspective, there are two interacting phenomena that contribute to this result: First, the memory latencies associated with pointer chasing in the linked lists appear to be partially masked by PForDelta de-

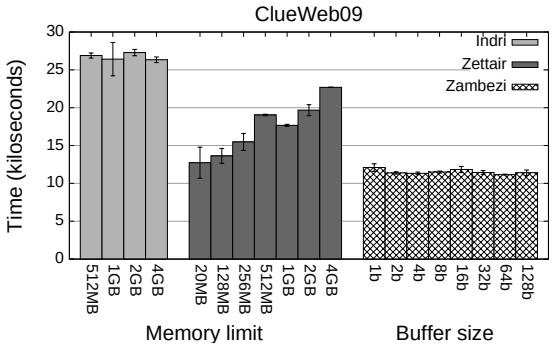

Figure 3: Indexing speed for Indri and Zettair with different memory limits, and Zambezi (our indexer) with different contiguity conditions. Error bars show 95% conf. intervals across 3 trials.

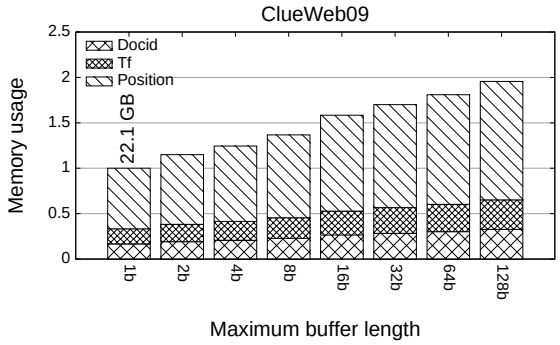

Figure 4: Memory required for all buffer maps with different buffer lengths, normalized to the $1b$ setting.

compression. With contiguous postings lists, predictable striding allows prefetching to hide memory latencies, but postings are traversed in "bursts" since after reading each segment the algorithm must decode the blocks. Thus, decompression can hide some of the effects of cache misses for discontiguous postings: while the processor is decompressing one segment, it can dispatch memory requests for the next (since the instructions are independent). Second, query evaluation is more complex than a simple linear scan of postings lists: SvS performs galloping search for intersection and WAND uses pivoting to skip around in the postings lists. This behavior creates unpredictability in memory access patterns and reduces opportunities for the prefetchers to detect striding patterns. To illustrate this, consider the difference between ideal performance and the $1b$ baseline condition: the performance gap is much smaller for WAND than for SvS. This makes sense, since at each stage, SvS intersects the current postings list with the working set: this implies greater cache locality, so we obtain a bigger performance boost with contiguous postings lists. On the other hand, WAND pivots from term to term and at each step may advance the current pointer by an unpredictable amount; even if the postings lists are contiguous, the processor may encounter cache misses. Thus, it makes less difference whether postings lists are contiguous to begin with.

5.2 Indexing Speed

Figure 3 shows indexing times for our indexer, Zettair, and Indri. For Zettair and Indri, we varied the amount of memory provided to the system. Note that we were not able to provide Zettair with more than 4GB memory due to its implementation. In C, the maximum size of an individual array is 2^{32} and circumventing this restriction would have required substantial refactoring of the code, which we did not undertake. For our indexer, we report results with different postings list contiguity conditions. Error bars show 95% confidence intervals across 3 trials. In all conditions we do not include document pre-processing time.

Indexing time with Indri appears to be relatively insensitive to the amount of memory provided, but it is overall slower than both Zettair and our indexer. With Zettair, the maximum size of the memory buffer does have a significant impact on indexing time. Surprisingly, giving Zettair more memory actually slows down indexing speed! We explain this counter-intuitive result as follows: smaller in-memory segments are more cache-friendly; for example, our system has a 12MB L3 cache, so in the 20MB condition, more than half of the segment will reside in cache. On the other hand, smaller segments require more merging. Overall, it seems that the first factor is more important: indexing is fastest with 20MB buffers.

These results show that our in-memory indexing algorithm is not substantially faster than an on-disk algorithm. Why might this be so? First, on-disk indexing algorithms have been studied for decades, and so it is no surprise that state-of-the-art techniques are well-tuned to the characteristics of disks. Second, cache locality effects and memory latencies slow down in-memory algorithms as the memory footprint increases—this is confirmed by the Zettair results, where, in general, giving the indexer more memory reduces performance. How does this happen? A larger in-memory footprint means that we are accumulating more documents in the buffer, and hence managing a larger vocabulary space. This causes more "cache churn", since whenever we encounter a rare term, its associated data (e.g., recently-inserted postings) are fetched into cache, displacing another term's. Since the rare term is unlikely to appear in another document soon, it is wasting valuable space in the cache. In contrast, the merging operations for the on-disk algorithms access data in a very predictable pattern, thus creating opportunities for the prefetchers to mask memory latencies.

Finally, we note that end-to-end system comparisons conflate several components of indexers that may have nothing to do with the algorithms being studied—for example, we use PForDelta compression, whereas Zettair does not. Thus, Zettair may be fast for reasons that are independent of the on-disk vs. in-memory distinction.

5.3 Memory Usage

All inverted indexing algorithms require transient working memory to hold intermediate data structures. For on-disk incremental indexing algorithms, previous work has assumed that this working memory is small. In our case, there is no hard limit on the amount of space we can devote to working memory, but space allocated for holding intermediate data takes away from space that can be used to store the final compressed postings lists, which limits the size of the collection that we can index for a fixed server configuration.

At minimum, our buffer maps must hold the most recent b docids, term frequencies, and associated term positions. In our case, we set $b = 128$ to match best practices PForDelta block size. Figure 4 shows the maximum size of the buffer maps for different contiguity configurations (as we increase

the buffer lengths), broken down by space devoted to docids, term frequencies, and term positions. The reported values were computed analytically from the necessary term statistics, making the assumption that all terms reach their maximum buffer sizes at the same time, which makes these upper bounds on memory usage. Since we primarily care about *relative* memory usage, we normalized the values to the $1b$ condition; in absolute terms, the total buffer map size is 22.1GB. It is no surprise that as the maximum buffer length increases, the total memory requirement grows as well. At $128b$ the algorithm requires 95% more space (compared to the $1b$ condition). At $32b$, which from our previous results achieves query evaluation performance that is statistically indistinguishable from contiguous postings lists, we require 70% more memory. As a reference, the total size of the segment pool (i.e., size of the final index) is 62GB. This means that setting the maximum buffer length to $1b$, $32b$ and $128b$ results in a buffer map that is approximately 32%, 54%, and 63% of the size of the segment pool, respectively. These statistics quantify the space overhead of our in-memory indexing algorithm.

Note that most of the working memory is taken up by term positions; the requirements for buffering docids and term frequencies are modest. The present implementation uses 32-bit integers in all cases, including term positions. We could easily cut the memory requirements for those in half by switching to 16-bit integers, although this would require us to either discard or truncate long documents.

The total number of unique terms is 79M in ClueWeb09. Since the collection consists of web pages, most of the terms are unique and correspond to JavaScript fragments that our parser inadvertently included and other HTML idiosyncrasies; such issues are prevalent in web search and HTML cleanup is beyond the scope of this paper. Our indexer discards terms that occur fewer than 10 times, which results in a vocabulary size of 6.9M words.

The average size of each inverted list segment for terms with a buffer length of $1b$ is about 300 bytes; for terms that require a buffer of length of $2b$, the average length is around 600 bytes. For terms with a buffer of length $> 2b$, this value is about 800 bytes. These statistics make sense since $1b$ terms may have a document frequency of less than 128, and in general, rarer terms have smaller term frequencies, and hence fewer term positions.

6. CONCLUSION

The finding in this paper that postings list contiguity matters only to a certain extent contributes to our understanding of main memory algorithms for text retrieval in the context of modern processor architectures. We believe that more work is needed to better understand the performance of query evaluation optimizations that may introduce different memory access patterns. One thing is clear though: the design tradeoffs for retrieval engines become very different once disk is removed from the picture.

7. ACKNOWLEDGMENTS

This work has been supported by NSF under awards IIS-0916043, IIS-1144034, and IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsor. The first author's deepest gratitude goes to Katherine, for her invaluable encouragement and wholehearted support. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

8. REFERENCES

[1] N. Asadi and J. Lin. Document vector representations for feature extraction in multi-stage document ranking. *Information Retrieval*, 16(6):747–768, 2013.

[2] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[3] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. *CIKM*, 2003.

[4] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. *VLDB*, 1994.

[5] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. *ICDE*, 2012.

[6] S. Büttcher, C. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. *SIGIR*, 2006.

[7] J. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM TOIS*, 29(1):1, 2010.

[8] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. *SIGIR*, 1990.

[9] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *VLDB*, 2011.

[10] R. Guo, X. Cheng, H. Xu, and B. Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. *CIKM*, 2007.

[11] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8):713–728, 2003.

[12] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. *CIKM*, 2005.

[13] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM TODS*, 33(3):19, 2008.

[14] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool Publishers, 2011.

[15] G. Margaritis and S. V. Anastasiadis. Low-cost management of inverted files for online full-text search. *CIKM*, 2009.

[16] M. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. *DASFAA*, 1997.

[17] W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *IP&M*, 41(2):275–288, 2005.

[18] K. Shoens, A. Tomasic, and H. García-Molina. Synthetic workload performance analysis of incremental updates. *SIGIR*, 1994.

[19] T. Strohman and W. Croft. Efficient document retrieval in main memory. *SIGIR*, 2007.

[20] T. Strohman and W. Croft. Low latency index maintenance in Indri. *OSIR Workshop*, 2006.

[21] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. *SIGMOD*, 1994.

[22] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. *WSDM*, 2013.

[23] H. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software–Practice & Experience*, 31(10):925–939, 2001.

[24] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. *WWW*, 2009.

[25] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(6):1–56, 2006.

[26] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. *ICDE*, 2006.