Efficient Immediate-Access Dynamic Indexing

Alistair Moffat
The University of Melbourne, Australia

Joel Mackenzie
The University of Queensland, Australia

Abstract

In a dynamic retrieval system, documents must be ingested as they arrive, and be immediately findable by queries. Our purpose in this paper is to describe an index structure and processing regime that accommodates that requirement for immediate access, seeking to make the ingestion process as streamlined as possible, while at the same time seeking to make the growing index as small as possible, and seeking to make term-based querying via the index as efficient as possible. We describe a new compression operation and a novel approach to extensible lists which together facilitate that triple goal. In particular, the structure we describe provides incremental document-level indexing using as little as two bytes per posting and only a small amount more for word-level indexing; provides fast document insertion; supports immediate and continuous queryability; provides support for fast conjunctive queries and similarity score-based ranked queries; and facilitates fast conversion of the dynamic index to a "normal" static compressed inverted index structure. Measurement of our new mechanism confirms that in-memory dynamic document-level indexes for collections into the gigabyte range can be constructed at a rate of two gigabytes/minute using a typical server architecture, that multi-term conjunctive Boolean queries can be resolved in just a few milliseconds each on average even while new documents are being concurrently ingested, and that the net memory space required for all of the required data structures amounts to an average of as little as two bytes per stored posting, less than half the space required by the best previous mechanism.

Keywords and phrases Information retrieval, web search, index construction, index compression, querying

Category Author Version. Please refer to the published version if you plan to cite this work: https://doi.org/10.1016/j.ipm.2022.103248

1 Introduction

An inverted index is a key component of most information retrieval (IR) systems. The standard inverted index structure consists of a *vocabulary* that maps strings to numeric term identifiers and also stores any required global information about the term (for example, the number of documents that contain it one or more times); plus a set of *postings lists* that record, for each term t, the set of documents that t appears in. Each postings list is a sequence of *postings* that allows any bag-of-words query Q to be resolved by accessing the $|Q|$ postings lists for the query terms, and then using the locational and occurrence information that they contain as input to a similarity computation. Zobel and Moffat [66] and Büttcher et al. [14] provide overviews of inverted index-based text querying. If the set of documents that is the target for such queries is fixed and stable, the index can be computed in a pre-processing phase, and then used during querying as a static resource. In this case there are two desired attributes which are likely to be in tension: the index should be as compact as possible, to minimize the storage footprint; and queries should be able to be executed as quickly as possible, to both minimize the computational footprint [17, 46] and to improve the user experience [5, 6].

On the other hand, if the set of documents is *dynamic*, with new documents arriving intermixed as part of a stream of operations that includes insertions as well as queries, an additional goal is introduced: ingestion of documents should be as fast as possible, and should be done in a manner that allows new documents to be immediately found in response to subsequent queries. That is, in a dynamic collection it is important that the index be continuously queryable, even as documents are being added. Figure 1 illustrates the resultant three-way operational tension, and thus the spectrum of possibilities for designing extensible indexing and query processing systems. Each of the three vertices represents a single optimization goal, with the open space between them showing the available span of three-way tradeoffs. For example, while it is possible to achieve fast insertion by regarding individual postings as elements in a linked list and appending them one-by-one into a single array of nodes [28], doing so is both expensive in terms of space, and also costly in terms of query speed. Similarly, compression techniques such as binary interpolative coding are known to obtain highly compressed representations (for example, see Pibiri and Venturini [52]), but they operate on whole postings lists, meaning that incremental updates would require expensive cycles of decompression and recompression.

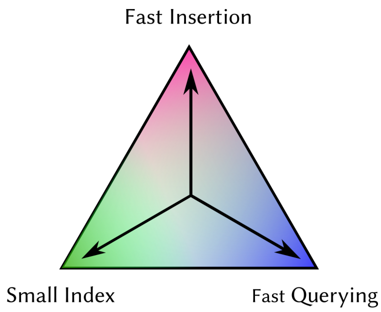

Figure 1 Trade-offs in the design of dynamic IR systems.

1.1 Goals and Contribution

In this paper we provide new trade-off points in the space defined by Figure 1. In particular, we describe (Section 3) a carefully engineered dynamic inverted index that balances the three performance dimensions shown in Figure 1 and provides fast ingest of new documents, a reduced storage footprint compared to previous dynamic indexing techniques, and efficient immediate-access querying. Our approach is able to support immediate-access dynamic indexing on typical document streams such as Wall Street Journal newspaper articles and Wikipedia articles using around two bytes per posting in a document-level index to cover all index costs (the vocabulary search structure, including the terms themselves and their associated global term information, plus all of the postings lists); and less than three bytes per posting if term-position information is also required, for example, to support phrase or proximity querying modes [16]. These compression rates mean that the index requires less than half of the storage space of the most recent dynamic immediate-access indexing method, that of Eades et al. [28]. The economy of space that we have achieved in part arises because of the innovative way in which we have structured the collection vocabulary as a component of the first block of postings for each term, and in part is a result of a novel byte packing operation that allows substantial reduction in the average cost of storing the postings. The new byte packing operation also has applicability beyond text indexing. As a final innovation, the structure of our new immediate-access index also allows for skip-links that allow fast conjunctive Boolean querying to be performed.

Our experiments (Section 4) show that our structure achieves an attractive balance of attributes, and that we compare very favorably with the dynamic index scheme of Hawking

Type	Postings	Explanation
document-level	$\langle d_{t,i}, f_{t,i} \rangle$	d-gaps $g_{t,i} > 0$ and corresponding frequencies $f_{t,i} > 0$
word-level	$\langle d_{t,i}, f_{t,i}, \hat{w} \rangle$	as above, and adding a list of w-gaps for word positions
word-level	$\langle d_{t,i}, w_{t,i,j} \rangle$	d-gaps $g_{t,i} \geq 0$ and w-gaps $w_{t,i,j} > 0$ within document $d_{t,i}$
word-level	$\langle w_{t,j} \rangle$	w-gaps $w_{t,j} > 0$, without regard to document boundaries

Table 1 Four different types of inverted index, in each case for some term t, with $d_{t,i}$ the i th document that contains t, and with $f_{t,i}$ the number of instances of t in $d_{t,i}$.

and Billerbeck [32], which offers similar ingest speeds, but builds indexes that are more than twice as large. Section 5 then considers a number of related issues, including word-level indexing, and algorithmic issues associated with extensible lists. In particular, we take a fresh look at the question of how many storage slots to allocate in response to an incremental request when the final total size required is not (and may never be) known. We show that a novel approach based on the triangle number sequence (rather than the previously-employed arithmetic or geometric sequences) has asymptotically smaller overhead wastage, and generates additional memory space savings.

Measurement of the new mechanism confirms that in-memory dynamic document-level indexes for collections into the gigabyte range can be constructed at a rate of two gigabytes/minute using a typical server architecture. Further, our experiments also demonstrate that multi-term conjunctive Boolean queries can be resolved in just a few milliseconds each on average even while new documents are being ingested as part of the same stream of operations, and that the net memory space required for all of the required data structures amounts to an average of as little as two bytes per stored posting.

2 Background

In this section we introduce a range of background material, including previous proposals for dynamic index structures, thereby setting the scene for the detailed development in Section 3.

2.1 Inverted Indexing

In a *document-level* inverted index each postings list contains postings of the form $\langle d_{t,i}, f_{t,i} \rangle$, with $d_{t,i}$ the ordinal identifier of the i th document containing term t, and $f_{t,i}$ the corresponding within-document frequency that counts the number of occurrences of t in $d_{t,i}$. It is usual to store the postings in *document order*, that is, sorted by increasing values of $d_{t,i}$, and for the original set of ordinal document numbers to be converted to a sequence of *d-gaps*, $g_{t,i}$, via the transformation $g_{t,1} = d_{t,1}$, and thereafter $g_{t,i} = d_{t,i} - d_{t,i-1}$. Conversion to gaps renders the postings lists more compressible (see Section 2.2), and we employ gap-based document numbers throughout this paper. In particular, even when we refer to a $\langle d_{t,i}, f_{t,i} \rangle$ tuple, what is actually stored is a $\langle g_{t,i}, f_{t,i} \rangle$ tuple. Note that $g_{t,i} \geq 1$ in all postings in a document-level index, since there is at most one posting per term per source document. Zobel and Moffat [66] and Büttcher et al. [14] further explain inverted indexing and provide examples.

In a *word-level* inverted index the location of each word in each document is also tracked. One way this can be done is via postings that include a third component $\hat{w}$ that contains a list of the $f_{t,i}$ ordinal word positions of t within that $d_{t,i}$ th document, themselves stored as a list of w-gaps relative to the start of that document. A second approach – and the option employed by Hawking and Billerbeck [32], and also the one we make use of in our work here – is to store tuples $\langle d_{t,i}, w_{t,i,j}\rangle$, where $w_{t,i,j}$ is the word position of the j th instance of term t within document $d_{t,i}$. The $w_{t,i,j}$ sequence across consecutive postings for any given document $d_{t,i}$ can again be reduced to w-gaps. However in this representation care must be taken with the d-gaps that form the first component of each posting, because now there might be multiple postings for t within a document d, meaning that $g_{t,i} \geq 0$ is the best that can be assured.

The third option for word-level indexing completely separates the d components from the w components, with each postings list simply a list of ordinal term positions in the entire collection, without explicit inclusion of document numbers. The resulting postings sequence $\langle w_{t,j}\rangle$, in which $w_{t,j}$ is the word position in the collection of the j th occurrence of term t, can be stored as a list of whole-of-collection w-gaps, to reduce the space required; this is the approach employed by Büttcher and Clarke [12]. The drawback of this third option – which is sometimes referred to as being a *schema-independent index* – is that when corresponding document numbers are required, they must be determined via binary (or other) search in an array that records, for each ordinal document number, its first word number in the collection. On the other hand, the schema-independent approach is the most compact of the three – placing it at a different location in the space of options described by Figure 1.

Table 1 summarizes the properties of these four types of inverted index. Our new implementation of immediate-access indexing explores the first and the third of the four possibilities shown. The choice between the three word-level options was in part determined by typical querying use cases, including the need for document boundaries to be distinguishable; and in part by the desirability of each posting containing two components, the benefit of which is explained in Section 3.4.

2.2 Index Compression

Compact storage of the postings list is essential, since they comprise the great bulk of any index, especially word-level ones. Fortunately, once ordinal values have been reduced to d- and/or w-gaps, a range of highly effective integer compression techniques can be applied, see Pibiri and Venturini [52] for a survey. In conjunction with document reordering techniques [3, 26, 42], the best of those integer coding techniques can typically represent document-level postings in less than eight bits each on average, a measurement we refer to as being the *compression effectiveness*. However, both document reordering and high-effectiveness integer coding regimes are based upon holistic analysis of the gapped values making up each posting list, an option that is not available in dynamic indexing applications.

Instead, dynamic indexes usually make use of *byte codes*, collectively referred to as being the VByte mechanism (although there are also small differences between specific descriptions). This form of code has been known for more than five decades. For example, Heaps [33] describes a general approach to integer compression that includes arrangements in which the code lengths are 8, 16, 24 (and so on) bits long; and Cutting and Pedersen [21] also describe a VByte mechanism. Williams and Zobel [65] include VByte in their experimental study, and further comparison was undertaken by Scholer et al. [54] and by Trotman [60, 61]. Subsequent developments are then reported by de Moura et al. [24], by Brisaboa et al. [7], by Culpepper and Moffat [19], by Dean [25], by Stepanov et al. [57], and most recently, by Lemire et al. [37].

Encoding non-negative integers $x \geq 0$ using VByte is straightforward, and operates quickly. At each iteration the low-order seven bits of x are isolated via a mask, and the remaining high-order bits are considered. If those high-order bits are zero, then the low-order bits are written as a byte with a 1-bit in the vacant top position, the latter indicating that x is now finished. On the other hand, if the high-order bits of x are non-zero, the low-order seven bits are written as a byte value between 0 and 127, and the process is iteratively applied to the value x' that contains the high-order bits of the original value x, shifted seven bits right. That is, each seven-bit segment of x is written within an eight-bit byte, with a "`0`" top-bit indicating "continue, another byte is required", and a "`1`" top-bit indicating "stop now, this is the last byte for this value". For example, the decimal number 12,345 requires fourteen bits in binary, "`1100000 0111001`", and spans two seven-bit segments. It would thus be written as two eight-bit bytes: "`0 0111001` + `1 1100000`", where the leading bits in each byte are the flag bits that indicate whether to continue or to stop. Decoding using `VByte` methods is also fast, involving mask and shift operations only.

Büttcher and Clarke [12] give pseudo-code for the `VByte` variant that we employ here. One key point to note in connection with this version is that there is only one possible cause of an all-zero byte in the compressed sequences – the pattern "`00000000`" (a null byte) can only arise as the code for an input value $x = 0$. This fact means that if we ensure that $x > 0$ at all encoding steps, then null bytes will never appear as part of a coded integer, allowing "`00000000`" to be used as a sentinel to mark the end of a sequence of compressed values, without the length of the sequence (nor its compressed length) needing to be stored explicitly. When a `VByte` decoder encounters a null byte, the compressed sequence has ended.

While `VByte` is a static code and thus needs no pre-analysis of the data that it is to be applied to, for document-level postings list compression it has the disadvantage of requiring a minimum of two bytes per posting, one for the $d_{t,i}$ value and one for the $f_{t,i}$ component (or for the $w_{t,i,j}$ component in a word-level index stored as $\langle d_{t,i}, w_{t,i,j}\rangle$ postings). Section 3.4 introduces a packing protocol that addresses that drawback and reduces the average cost of $\langle d_{t,i}, f_{t,i}\rangle$ postings for typical document collections to approximately 1.5 bytes per posting, allowing a substantial space reduction to be achieved.

Many other highly-effective compression methods exist. But they typically act holisticly on blocks of postings [36, 50, 52, 64], thereby creating a tension between storage space and updateability, as has already been noted in the trade-off spectrum shown in Figure 1. Static compression techniques are the methods of choice for immediate-access indexing.

2.3 Static Index Construction

Early index construction techniques were designed for static collections, and operated as two or more processing phases, often also relying on external disk storage. For example, Fox and Lee [30] describe a multi-pass arrangement in which each traversal of the input text builds the postings lists for a subset of the terms, with the number of passes dictated by the relationship between the total number of postings in each fragment of the index, and the amount of main memory available to store them while the lists are constructed.

Harman and Candela [31] describe an in-memory inversion process, using a binary search tree to store the vocabulary of the collection, with a linked list of postings attached to each tree node to thread together the term observations. At the end of the input the index is recovered by traversing the tree in term order, and at each node, following the list links. In a different approach, Moffat [47] showed that for one type of efficient postings list compression method the cost of storing the postings lists in compressed form could be upper-bounded based solely upon knowledge of the term frequencies, and described a two-pass inversion process that first builds a collection vocabulary and counts term frequencies; then allocates the starting points of the postings lists of a compressed inverted index in memory; and then, in a second pass, computes the d-gaps and $f_{t,i}$ values, and fills in the compressed index in a random-access order. While still limited by the amount of main memory available, the use of effective compression meant that relatively large amounts of text could be efficiently handled.

In subsequent work Moffat and Bell [48] remove the limit imposed by main memory, while still using compression as an important part of the process. In this approach incoming documents are processed in batches to accumulate sets of posting that fit within the available memory limit, and when that limit is reached, the postings are sorted and a local index for that document batch is written to disk in compressed form. When all of the documents have been ingested, the set of partial indexes is combined via a multi-way sequential in-place on-disk merge operation that maintains – and wherever possible, improves – compression effectiveness, so as to operate within the same envelope of disk space as was consumed by the set of compressed partial indexes. The method of Moffat and Bell can thus be regarded as being complementary to the earlier work of Fox and Lee [30] – the latter manage large collections by splitting the vocabulary and making multiple passes, whereas Moffat and Bell recommend instead that the collection be partitioned on a "by documents" basis.

Heinz and Zobel [34] added further enhancements. They observed that there is no requirement for a comprehensive vocabulary covering all terms to be constructed until the end of the inversion process, and that the vocabulary can thus also be created in batches. In their proposal batches of documents are again inverted using as much of the available memory as can be made available, maximizing the number of documents in each batch so as to stay within that memory limit. They also directly build localized posting lists using compression, rather than accumulating raw postings for later sorting, increasing the number of documents handled in each batch, and hence reducing the number of batches required. Once constructed, each partial (compressed) index is written to disk, including its own local vocabulary. Then, when all of the document batches have been processed, the set of local vocabularies is merged to make a whole-of-collection vocabulary; and the terms' postings lists are read, interleaved as required, and then written. Only one pass is made over the source document collection, and although there is a second processing phase to carry out the merge, it operates over data that has already been processed into binary format, and is not as expensive as a full pass over the source text would be.

2.4 Query Processing Using Static Indexes

Static indexes naturally lend themselves to efficient querying algorithms, with a wide range of such techniques having been developed. One of the keys for achieving efficiency over static inverted indexes is to embed "skip pointers" into the index at the time it is constructed [51], so that runs of postings can be quickly bypassed if they are not required by the current query, thus saving both storage accesses and potentially costly decoding operations. Skip pointers are critical for efficiently supporting fundamental querying modes such as Boolean conjunctions [20, 35].

Disjunctive query modes, which return the top-k documents based on some similarity estimation calculation, also benefit from static indexes. By employing a combination of skip pointers, encoded document scores within postings lists, and a query-time data structure to record the best scores "seen so far", it is possible to bypass documents known to have no prospect of being among the final top-k results. These approaches, known collectively as *dynamic pruning*, have been a focus of attention through several decades [8, 23, 27, 43, 63]. We refer the interested reader to the surveys of Tonellotto et al. [59] and Mackenzie and Moffat [41] for further details of these.

2.5 Dynamic Indexing and Querying

Each in-memory index assembled during the operation of the Heinz and Zobel [34] approach is ostensibly one step towards a global index for a large document collection. But at any given instant the in-memory component can be configured to provide the functionality of an operational inverted index for the current batch of documents, and any queries that might arise can be processed against the current document batch in the normal manner.

As part of their index, Heinz and Zobel [34] maintain postings lists using *extensible arrays*, one per term. Extensible arrays are a well-known data structure: when an array of unknown eventual size is required, it is allocated $n_1 = 1$ cells at first; thereafter, at each subsequent insertion, if the array is full and all n_z cells are occupied, a new array of size $n_{z+1} = \lceil k \cdot n_z \rceil$ is allocated for some $k > 1$ (with $k = 2$ being a typical value); the contents of the old array are copied to the new one; the space associated with the old array is released; and then the process is continued, now with $n_{z+1} - n_z - 1$ empty slots in the new array (after the one that triggered the expansion got claimed). The geometric sizes determined by the multiplicative growth rate k mean that when the array has reached n elements the total number of item copies over all of the completed growth cycles is bounded above by $nk/(k-1)$; that is, the total time required to sequentially append n elements to an initially empty array is $O(n)$. However extensible arrays are not so efficient in terms of space. At each expansion moment an extensible array of size n_z is fully occupied, and another array of size n_{z+1} has been allocated and is awaiting its contents, at a time when $n = n_z + 1$. That is, there are repeated instants at which a total of $(2 + k)(n - 1) \geq 2n$ cells are allocated to the storage of n elements.

Büttcher and Clarke [12] adopt much of the approach of Heinz and Zobel, but with one critical difference – they make use of *extensible lists*, in which blocks of items are linked together using pointers, with each postings list stored as a chain of such blocks. They argue that the overhead associated with maintaining one pointer per block, plus having a small amount of space unused in the last block, is a more attractive compromise than the multiplicatively-growing unused space incurred by an extensible array. They further argue that during querying the need to follow a pointer from time to time is an acceptable overhead.

Büttcher and Clarke then go on to consider two strategies for setting the block size: in the Const_B approach, each block is B words, with one word required for the pointer to the next block, and $B - 1$ words available for *payloads*, that is, $B_{z+1} = B_z = B$. In the alternative $\mathsf{Expon}_{B,k}$ approach, the payload capacities grow as a geometric sequence, $B_{z+1} = 1 + \lceil k \cdot (B_z - 1) \rceil$, with $B_1 = B$, and the assumption again being that each pointer requires the same space as one payload element. At any given moment a chain of z blocks thus contains up to $n = \sum_{i=1}^{z}(B_i - 1)$ payloads, and z pointers. Büttcher and Clarke [12] experiment using both synthetic Zipfian data and also two different TREC collections, and conclude that the $\mathsf{Expon}_{B=16,k=1.1}$ method, modified by the addition of an upper cap of 256 on block size, provides the best balance in terms of indexing memory and indexing throughput. With that approach they are able to obtain $\langle w_{t,j} \rangle$-format word-level indexes (see Table 1) that require around 1.8 bytes per posting on typical TREC document collections, not including the vocabulary and term structure costs.

Asadi et al. [4] and Busch et al. [10] also consider the challenge of dynamic indexing and immediate-access search, with an emphasis on the requirements of the Twitter search service. They note that tweets are often required in reverse chronological order in response to Boolean searches, and hence employ uncompressed postings stored as 32-bit words, reverse-chaining segments of size 2^1, 2^4, 2^7, and 2^{11} postings.

In other recent work, Hawking and Billerbeck [32] revisit the mechanisms proposed by Büttcher and Clarke [12], exploring further blends of Const and Expon extensible lists, and considering the way that those strategies interact with the non-linear behaviors introduced by specific hardware characteristics. Hawking and Billerbeck make only limited use of compression, and the most compact index they obtain for a $\langle d_{t,i}, w_{t,i,j}\rangle$-format word-level inverted index against a collection of 8.46×10^9 web pages requires around 6.2 bytes per posting, again not including the vocabulary and hash table costs.

2.6 Other Related Work

The in-memory techniques of Heinz and Zobel [34], Büttcher and Clarke [12], and Hawking and Billerbeck [32] all yield indexes for a batch of documents. Other work has investigated how best to join such partial indexes together. For example, Lester et al. [38] propose that each index batch be immediately merged with a single on-disk index. This approach results in growing merging costs as the two components of each merge become increasingly disparate in size, and is not asymptotically efficient; but also has the advantage that any given query requires only two collections to be searched, the main on-disk index, and the current in-memory batch index.

In contrast, Lester et al. [39] (see also Büttcher and Clarke [11]) propose a hierarchical merging strategy that always joins components of approximately equal size and is asymptotically efficient, but requires that queries be processed against both the current in-memory index and a logarithmic-bounded number of on-disk indexes, with the base of the logarithm determined by the batch size possible. Büttcher and Clarke [13] consider a range of hybrid strategies that blend these two approaches. The work we describe in this paper can be used as a first processing phase and combined with any of these memory-to-disk merging strategies.

Earlier work also considered dynamic index construction and querying operations. For example, Cutting and Pedersen [21] consider the batching of updates into an inverted index stored as records in a B-tree structure, and show that it is better to use any extra memory to accumulate postings into batches than it is to cache upper-level B-tree nodes; they also consider the case when postings must be displaced from main memory on to disk. The work of Faloutsos and Jagadish [29] is also directly relevant here: they consider extensible arrays, extensible lists, and also a hybrid that combines them, in each case regarding the set of postings as the leaves of a B-tree. They also explore several of the growth strategies that have already been mentioned, including Const and Expon, analyzing their relative performance against a Zipfian distribution of occurrences, and measuring storage effectiveness (but not employing any form of compression) and search times using datasets derived from the 11,657 first and family names extracted from a university phone directory.

Shoens et al. [56] and Tomasic et al. [58] also examine Zipfian distributions, and suggest the use of two different types of postings list elements, short ones that store only a small number of postings each, but also reduce the risk of excessive unused space being associated with "tail" terms; and long ones with a greater postings capacity, to be allocated when sufficient evidence of demand for that term has been observed. As documents are received and postings accumulated, terms are dynamically shifted from the short-block category to the long-block category if they exceed their initial space allocation, or if other terms arise that pressure them out of their current block.

Brown et al. [9] also describe dynamic indexing as being a process in which objects (postings lists) grow and must be extensible. At any given moment each posting list is one of a small palette of possible sizes (16, 32, 64, through to 8192 bytes), with long lists that exceed 8192 bytes handled by chaining blocks together, another hybrid scheme. Clarke et al. [18] also make use of batch update operations to extend the on-disk index needed when the collection is too large for its index to fit into main memory.

Extensible arrays and extensible lists incur "tail wastage" that occurs whenever the last allocated block is only part full. Shieh and Chung [55] seek to address that issue by monitoring the way in which lists accumulate postings, and employ statistical prediction techniques in connection with extensible arrays. They demonstrate higher space utilization and reduced relocation levels compared to the $\mathsf{Expon}_{2.0}$ and $\mathsf{Expon}_{1.5}$ mechanisms used as baselines. The new approach to extensible lists that we present in Section 5.4 has the benefit of reducing the tail wastage to an arbitrarily small fraction of the stored content as list sizes become larger, without needing to employ statistical estimation mechanisms.

In more recent work, Eades et al. [28] consider dynamic indexing from a different perspective. They address the question of *sliding window retrieval* in which recent documents must be queryable, but older documents are to be "forgotten". Their proposal, dubbed the *apoptosic index*, makes use of a circular array of n elements to retain the most recent n postings, with older postings expiring as new ones are received. Each posting contains four items, $\langle d, t, f_{t,d}, p\rangle$, where p is the position in the circular buffer of the previous instance of term t, and hence provides a back-pointer that can be used as a linked list during querying to step through the postings associated with t. In this arrangement – with careful attention paid to relative pointer positions – expired postings are simply overwritten, without a specific deletion operator being required. That is, postings are replaced on a one-for-one basis, meaning that if a certain number of documents are to be retained in the index, a maximum ratio of postings to documents must be known in advance. Eades et al. report experiments using an implementation in which the four posting components are represented as integers, with a set of n postings requiring $16n$ bytes (not including the cost of the vocabulary or term search structure), a non-trivial requirement when n is large. Eades et al. observe that compression can be applied to their postings, claiming that "16–20 bits per node [posting] should be straightforward". But with each back-pointer spanning a distance of one or more multiples of the average number of distinct terms per document (that is, many hundreds of positions in the circular array), that statement seems optimistic. Nor can the apoptosic method be used as a stepping stone in the construction of a conventional on-disk inverted index.

Finally in this section, we note the possibility of the appended documents not being incrementally indexed at all, and instead being searched in text format using sequential pattern-matching approaches. While this approach is easy to implement, the batch of inserted documents must eventually be indexed if the overall system is to be long-term efficient, and so deferred indexing is only plausible in highly restricted circumstances, such as when each document batch is to be filtered before indexing, with only a subset of them needing to be permanently retained.

3 Compact Immediate-Access Indexing

We now present our proposed structure, supposing that a stream of incoming documents is to be converted to a document-level inverted index (see Table 1; word-level indexes are discussed in Section 5.1), and that any new document must be instantly findable via a concurrent query stream. The key elements to note in this section are the role that immediate-access indexing plays within a larger system in which some parts of the index have already been reorganized into highly efficient static arrangements (Section 3.1); our use of fixed-length postings blocks,

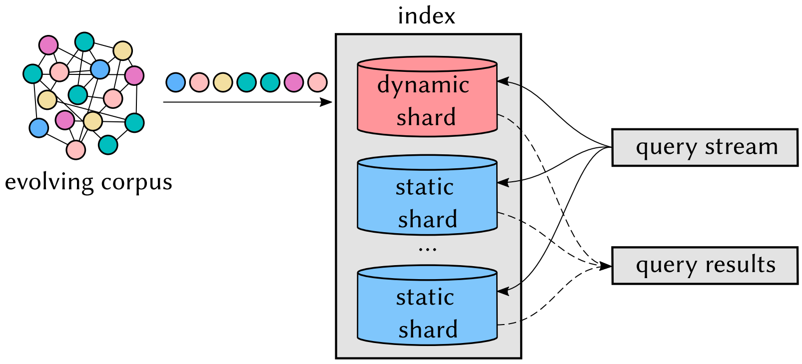

Figure 2 Overall structure of a large-scale search system in which documents are ingested and then permanently retained. An in-memory dynamic shard index accumulates arriving documents, and from time to time is shifted to secondary storage and reorganized to form a further static shard index. Each incoming query is processed against the dynamic shard index and all previous static shard indexes, and the results fused. The dynamic shard and static shards might have different index organizations, and might employ different query processing heuristics. Our focus in this paper is on the dynamic shard index, and the operations needed to ingest documents and to query against them.

the first of which also includes all of the vocabulary information (Section 3.2); the chaining together of the blocks for each term as documents are inserted (Section 3.3); the development of a packed Double-VByte compression mechanism that brings a dramatic reduction in the average cost of compressed postings (Section 3.4); and the preliminary compression and throughput results showing how effective it is (Section 3.5).

3.1 Mode of Operation

Figure 2 illustrates the role that an immediate-access dynamic index plays in a larger retrieval system. Ingested documents are immediately added to the dynamic index and rendered findable, a process that continues as long as the dynamic shard index can be retained in main memory. At the same time, previously accumulated document batches have had their index data reorganized (and possibly merged) so as to maximize retrieval speed and minimize stored space, thereby occupying a point on the bottom edge in Figure 1; they can be searched using the efficient querying techniques summarized in Section 2.4. When the current dynamic shard index has reached the available memory limit it too is converted to the same static form, to make it smaller and allow faster querying modes to be used. After being reorganized it is added – perhaps after being merged with other static index components – to the part of the index held in secondary storage. A new dynamic index is then initiated.

Because it faces different operational requirements, the dynamic shard has a different internal index structure, and occupies a different location in Figure 1. Incoming queries are resolved against all shards, using the corresponding strategy for the static and dynamic components, and fused answer sets then returned to the users or passed to a more sophisticated ranking mechanism. Our focus in this paper is exclusively on the construction and querying of the dynamic shard index, as a critical component of the larger search ecosystem illustrated in Figure 2.

Given this context, our research goals can now be stated

- How should a dynamic index be organized so as to best balance insertion cost, storage cost, and querying costs?
- To what extent do those necessary compromises erode compression and retrieval performance relative to state-of-the art static indexing and querying arrangements?

As a result of exploring those goals, and relative to the background established by the various methods summarized in Section 2, we are able to offer the following points of distinction:

- We describe an immediate-access indexing scheme that provides for ingest of new documents at speeds comparable to those of Hawking and Billerbeck [32];
- At the same time, use of a new compression approach means that our approach can process documents batches of roughly twice the size as Hawking and Billerbeck [32] within the same amount of memory; plus
- We document querying speeds in the immediate-access index, and while these are (unsurprisingly) slower than can be attained in a fully-optimized static retrieval system, they are neverthess fast enough that the overall static/dynamic structure shown in Figure 2 becomes practical.

3.2 Fixed-Block Indexing

As do Büttcher and Clarke [12] and Hawking and Billerbeck [32], we make use of chains of blocks, one per indexed term, with each block containing compressed postings. But compared to those methods there are also some key differences. First is that the vocabulary information is stored as a component of the first block associated with each term, rather than in a separate data structure, to reduce the overheads associated with both the variable-length term information and also with the first few postings. Packing these two variable-length entities together replaces and improves upon the earlier suggestion of Trotman et al. [62] that one or two postings be stored in the vocabulary. Second, we assume – in this section, at least – that all blocks are of the same length (the Const strategy) and include an $h = 4$-byte linking pointer within a B-byte total size; but also consider a novel variable-length block alternative in Section 5.4. With fixed-size blocks rather than a sequence of lengthening blocks, the index as a whole can thus be thought of as an array of blocks, with array offsets used to locate blocks within that array, rather than requiring byte-addressed pointers. In turn, this cuts down on the overhead cost of the meta-data needed to manage the index, and permits higher data loadings to be achieved. And third, we introduce (as an orthogonal improvement) a new way of compressing postings that reduces the cost of storing them by as much as one third compared to the usual VByte mechanism.

In particular, by maintaining just one block size, but employing around half of each term's first block to contain the vocabulary information and leaving half (or even less) for the first few postings, we in effect have two different block sizes in operation: short first postings blocks, and thereafter constant-length normal blocks, both containing exactly B bytes in total. That is, in this new arrangement we gain the same benefit as in previous approaches that used postings blocks of increasing capacities, without the complexity of needing to manage a range of memory allocation amounts.

Figure 3 illustrates these ideas, showing the chain of blocks associated with one term t, noting the three types of blocks that arise (*head*, *full*, and *tail*) and showing the fields associated with each. The head block is located via a hash table, and contains all the standard vocabulary information as well having the capacity to store a modest number of postings. Subsequent blocks of postings are then linked by pointers (denoted as n_ptr in the diagram), and by a single tail pointer (shown as t_ptr) that allows direct access from the head block to the tail block. The latter is normally only partially full, and is the site of

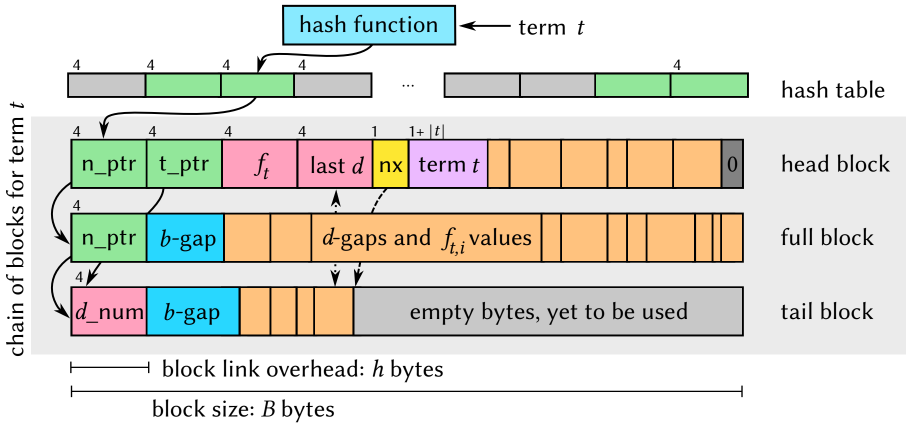

Figure 3 The chain of B-byte blocks associated with one term t, showing the three types of blocks that might arise and the way in which the vocabulary (global whole-of-term information) and postings (localized information) are distributed across them. The superscript numbers besides some of the fields indicate the length in bytes of those elements. The d-gaps, $f_{t,i}$ values, and b-gaps are all variable length codes. Any unused trailing bytes are assigned the value character null (as shown at the end of the head block).

any update operations for this term, with the field nx in the head block indicating the offset in the tail block of the current "write" location as that block gets filled. Term t's blocks are interleaved with the blocks of other terms (not shown in Figure 3) in a single large array of fixed-length blocks which we denote as $\mathcal{I}$. Each postings list contains one head block; zero or more full blocks; and one tail block, which is initially also the head block; overall, the index contains one head block for each distinct term.

All blocks store postings $\langle d_{t,i}, f_{t,i} \rangle$ coded using VByte (or as we shall describe shortly, a more efficient variant of it). In addition, the first document number in each full block is stored as a b-gap (block gap) rather than as a d-gap, computed as the difference between this block's first document identifier and the first document number in the previous block in the chain. The b-gaps allow an indexed sequential access mode that supports $seek\_GEQ(d)$ operations that scan a list for a given document d, touching only the b-gap and n_ptr during the scan. That is, the b-gaps provide support for *skipping*, first proposed by Moffat and Zobel [51].

With the vocabulary stored as a part of the index $\mathcal{I}$, the process of determining the correct head block for any given term must also be handled carefully. We employ a hash array of 32-bit integers that stores block offsets, and map the characters of each term t via that array to the offset in $\mathcal{I}$ of t's head block. We assume a hash array twice the size of the collection vocabulary (using an extensible hashing technique), which then allows use of a simple linear advance collision resolution technique, and thus provides $O(|t| + 1)$-time search, where $|t|$ is the number of characters in t. That is, if v is the vocabulary size of the collection, the hash array is costed at $8v$ bytes in our memory consumption results. All of the compression results given in Section 4 include *every* component of the index structures required.

As noted in Figure 3, we make use of $h = 4$-byte block counts, limiting $\mathcal{I}$ to 2^{32} blocks. With $B = 64$ a typical block size, that means that our in-memory indexes are capped at 64×2^{32} bytes, which is 256 GiB. The net implication of that limit is discussed further at the end of Section 4.4.

Algorithm 1 Adding a posting $\langle d, f \rangle$ for some term t for which the head block $H \equiv \mathcal{I}[h\_ptr]$ has been identified by the hash-based mapping. Variable *nblocks* is a global count of the number of blocks in $\mathcal{I}$ that are in use; and each n_ptr value requires $h = 4$ bytes.

```
function add_posting(I, h_ptr, d, f):
    let H represent the block I[h_ptr]                         ▷ head block
    let T represent the block I[H.t_ptr]                       ▷ current tail block
    set gap ← d − H.last_d                                     ▷ compute d-gap
    set nbytes ← code_len(gap, f)                              ▷ do a test encoding
    if H.nx + nbytes > B then                                  ▷ and check for fit
        // need to allocate a new tail block, and convert previous tail block to “full”
        set gap ← d − T.d_num                                  ▷ use a b-gap at start of block
        now let F represent the block I[H.t_ptr]
        and let T represent the block I[nblocks]
        write null bytes from F[H.nx] to F[B − 1] inclusive
        set T.d_num ← d                                        ▷ note first-in-block docnum
        set H.t_ptr ← F.n_ptr ← nblocks                        ▷ set block pointers
        set H.nx ← h                                           ▷ set the VByte write pointer
        set nblocks ← nblocks + 1
        set nbytes ← code_len(gap, f)
    code gap and f as bytes into T, starting at T[H.nx]
    set H.nx ← H.nx + nbytes                                   ▷ advance the byte pointer
    set H.last_d ← d                                           ▷ note most recent docnum
    set H.ft ← H.ft + 1
```

3.3 Adding Documents

As new documents arrive they are parsed into terms, and repeated occurrences within the document collected together via a sort-counting process. Each term t is also mapped via the hash array and using the vocabulary information, to obtain the offset in $\mathcal{I}$ of t's head block.

If the document contains any terms t that have not appeared in previous documents, an empty head block is allocated for each, by taking the next unassigned block in $\mathcal{I}$, indicated by a single global counter that we denote as *nblocks*. At the same time, *nblocks* gets stored into the position in the hash table corresponding to t. New head blocks H then get their vocabulary components assigned, and have their tail block byte offset write counters $H.nx$ initialized to $4h + 2 + |t| = 18 + |t|$, the first location in H that will be used for postings bytes.

The list of unique terms associated with the new document and their corresponding head block offsets is then processed via a sequence of *add_posting* operations, one per term, each of which appends one posting to the index. The process for inserting one new posting is shown in Algorithm 1, with B the uniform length of each block in index $\mathcal{I}$. In the pseudo-code, variable $H \equiv \mathcal{I}[h\_ptr]$ corresponds to the head block for term t, identifying the term this posting is to be associated with; variable T is the current tail block and then the new tail block, should growth be required; and variable F denotes the previous tail block in those cases in which growth gives rise to the need for a new tail block. Either or both of T and F might indicate the same block as does pointer H.

The pseudo-code in Algorithm 1 treats all of H, F, and T as being both compound structures (or rather, as pointers to structures) and, when required, as plain arrays of B bytes. In particular, the "." operator is used to select the fixed-width elements at the front

Algorithm 2 Double-VByte, packing the gap g and frequency f for a posting into a single byte when $g < 128/F$ and $f < F$, for threshold parameter F.

function $double\_vbyte\_encode(g, f)$:
if $f < F$ **then** ▷ when the value f is sufficiently small,
  set $g' \leftarrow (g-1) \times F + f$ ▷ g and f can be combined into a single value g',
  $vbyte\_encode(g')$ ▷ and jointly coded using as few bytes as necessary
else
  set $g' \leftarrow g \times F$ ▷ or, when f is larger than the threshold,
  $vbyte\_encode(g')$ ▷ an inflated version g' of g is first coded,
  $vbyte\_encode(f - F + 1)$ ▷ followed by a slightly adjusted version of f
return

function $double\_vbyte\_decode()$:
set $g' \leftarrow vbyte\_decode()$ ▷ fetch a single value g' from the stream,
if g' **mod** $F > 0$ **then** ▷ and if it is not a multiple of F,
  set $g \leftarrow 1 + g'$ **div** F ▷ it carries both g and f embedded within it
  set $f \leftarrow g'$ **mod** F
else
  set $g \leftarrow g'$ **div** F ▷ or, when g' is a multiple of F, it contains g only
  set $f \leftarrow F + vbyte\_decode() - 1$ ▷ and f was coded as a separate element
return (g, f)

of an indicated block (see Figure 3), with byte-by-byte access operations past those fields carried out using the subscripting operator "`[]`", counting bytes from the beginning of the block. For example, with the first four fields in the head block H each stored as $h = 4$-byte integers (see Figure 3), the one-byte element $H.nx$ could also be referred to as $H[16]$. Note also that each block contains an integral number of postings, and that our implementation does not split VByte values across blocks – the integrity of the b-gap support for indexed sequential access was felt to be more desirable than this last small saving. Unused tail bytes that get created by this decision are set to null, allowing the VByte decoder to know that it has reached the end of the block, as discussed in Section 2.2.

Working through Algorithm 1 in detail, there are three phases to note. In the first phase, lines 4 and 5 compute the d-gap associated with the new posting, and then calculate the length of its compressed representation. The second section of code, lines 6 to 16, tests whether that posting can fit in what remains of the current tail block (step 6), and if it cannot, closes off the current tail block by writing null bytes in all unused positions (step 11); allocates a new block and sets the appropriate pointers (step 13) including the within-block byte counter (step 14); computes the required b-gap (step 8); and finally recalculates the compressed posting size (step 16). The third phase, at steps 17 to 20 then writes the posting (as a b-gap rather than a d-gap, if this is the first posting in the block) into the tail block, and finally adjusts three variables in the head block, thereby completing the *add_posting* operation and returning to the stable configuration shown in Figure 3.

3.4 Double VByte – Better Postings Compression

Algorithm 2 describes another important part of our proposal, a code-packing technique that can be used in any situation in which byte codes are used to store postings. The key idea is that when the $f_{t,i}$ value is small (defined in the pseudo-code as being less than the fixed value F, with $F = 4$ a typical value), instead of coding it independently and hence consuming at least one byte to store each of the $d_{t,i}$ and $f_{t,i}$ values, the two components are folded together into a single value before being coded. The folding operation can be unambiguously reversed at decode time; and is structured so that calls of the form *vbyte_encode*(0) can never arise, preserving the decoder's ability to know when to stop decoding, as noted in Section 2.2. The comments embedded in the right-hand side of Algorithm 2 provide further step-by-step guidance that explains the details of the Double-VByte encoding and decoding processes. For example, when $F = 4$, $g = 10$ and $f = 3$, a single byte covers the packed value $g' = (g - 1) \times F + f = 39$, and the posting is one byte shorter than had two separate VByte codes been used, one for g and one for f. Then, when decoding, the value $g' = 39$ is not a multiple of $F = 4$, and so the two components $g = 10$ and $f = 3$ can be extracted out of that same single byte.

The Double-VByte code relies on there being a majority of cases in which the computed value g' is small enough that a one byte VByte code suffices. The combined value might also become large enough that a VByte code of two bytes is needed, which is still no loss; this is what happens if $g = 40$ and $f = 3$, with $g' = (g - 1) \times F + f = 159$ needing two output bytes. The pigeonhole principle means that there must also be cases in which the $d_{t,i}$ and $f_{t,i}$ values would require one byte individually, but in conjunction end up requiring three bytes. An example of this third situation occurs when $g = 40$ and $f = 5$, with $g' = 160$ needing to be coded (two bytes) followed by $f - F + 1 = 2$ in a third byte.

3.5 Preliminary Results

Fortunately, the typical frequency distributions associated with postings data – a very high fraction of low values of f; many small values of g; and a joint distribution that has larger values of g highly likely to be accompanied by low values of f – mean that the third of those three cases is relatively rare and that it is the first case that dominates. Table 2 provides evidence in support of that claim, derived from the WSJ1 collection of approximately 100,000 documents and 21 million postings (see Table 5 in Section 4.1 for details of the three test collections used in our experiments). It shows the breakdown of postings costs, categorized by initial size across the columns as the sum of two separate calls to *encode_vbyte*(), and then by transformed size via a single call *encode_double_vbyte*(g, f) within each of the columns. The percentages in each column below the line add up to the number in the "before" row above the line, with all percentages relative to the total number of postings. The overwhelming dominance of the blue values (one byte saved) compared to the corresponding red values (one byte extra required) demonstrates the usefulness of the new technique. In this collection there were no five-byte postings required using either mechanism.

Table 3 translates those behavioral patterns into concrete costs, expressed now in units of bytes per posting (and in this preliminary table, counting postings only, and not including the hash array, nor any of the fixed costs associated with the index blocks). When $F = 1$ the original "separate VByte codes" scheme is in operation; as can be seen, relative to that baseline fully one third of the postings cost can be eliminated by Double-VByte when $F = 4$. We use $F = 4$ in all subsequent experiments on document-level inverted indexes.

We also carried out a preliminary "straight through" speed test experiment, to compare VByte and Double-VByte encoding and decoding rates. Table 4 summarizes the outcomes. In this experiment, the complete set of postings of the Wikipedia test collection (see Section 4.1 for details) was placed into an array of 32-bit integers in memory, as a sequence of alternating d-gaps and $f_{t,i}$ values, without any further metadata of any sort. They were then encoded

New size (Double-VByte, bytes)	Original posting size (VByte, bytes)			
	2 82.19%	3 16.82%	4 0.99%	5 –
1	58.51%	–	–	–
2	22.92%	14.57%	–	–
3	0.76%	2.18%	0.93%	–
4	–	0.08%	0.05%	–

Table 2 Percentages of $\langle d_{t,i}, f_{t,i} \rangle$ postings of each given length in the WSJ1 collection when represented as separate VByte codes for the d-gap and the $f_{t,i}$ component, and the distribution of Double-VByte sizes when jointly coded via Algorithm 2 using $F = 4$. Table 5 provides details of the document collection.

	$F = 1$	$F = 2$	$F = 4$	$F = 8$	$F = 16$
Bytes/posting	2.188	1.579	1.456	1.501	1.603
Ratio	1.000	0.722	0.666	0.686	0.733

Table 3 Double-VByte postings costs measured in bytes per posting and relative size ratio for the WSJ1 collection, as a function of transformation parameter F. When $F = 1$ the original VByte scheme results.

to a second array of unsigned bytes, measuring the time taken. That second array was then decoded back to a third array of 32-bit integers also in memory; and finally, the first and third arrays compared, to verify the integrity of the process. A parameter of $F = 4$ was used in Double-VByte.

As can be seen from the table, the Double-VByte mechanism is approximately 20% slower than is VByte, for both encoding and decoding; with the differential caused by the need for additional control logic, including an if-statement and possible branch (see Algorithm 2). In turn, both are slower than plain copy operations, in which bytes or whole words are transferred via a loop. In the last line of the table the standard function `memcpy()` is used, which is about three times faster than are VByte and Double-VByte. The right-most column of Table 4 provides the justification for the use of VByte and, even better, Double-VByte, confirming that relative to plain integers, Double-VByte saves almost 80% of the postings cost.

3.6 Querying Operations

The index structure shown in Figure 3 is always available for querying, and search operations can be interleaved with document insertions. To resolve a query Q the hash array and term strings stored in the corresponding entries in $\mathcal{I}$ are used to locate the set of active head blocks, one per query term, indicated by a set of h_ptr variables. Each of those head blocks contains the first tranche of postings for that term, with more blocks linked via the n_ptr fields. Within any given block posting decoding ends if a null byte is detected, or if the Bth byte has been reached, at which time the next block is accessed via that current block's stored n_ptr. Decoding of the entire posting chain for any term ends when the block

Method	Encoding	Decoding	Effectiveness
VByte	2.844	2.981	2.304
Double-VByte	3.451	3.555	1.629
Copy, looping over bytes	1.680	2.149	8
Copy, looping over integers	1.282	1.135	8
`memcpy`, whole slab at once	1.347	1.145	8

Table 4 Array-based encoding and decoding speed, measured over the 996,277,511 document-level postings of the form $\langle d_{t,i}, f_{t,i}\rangle$ extracted from the Wikipedia test collection; that is, approximately 1.99 billion integers. The Double-VByte parameter F was set to four; en/decoding times are measured in seconds for the complete set of postings (smaller is better); and compression effectiveness is measured in compressed bytes per posting (smaller is better). The experimental hardware is described in Section 4.1.

$\mathcal{I}[h\_ptr].t\_ptr$ is reached, and then, within that tail block, when the first $\mathcal{I}[h\_ptr].nx$ bytes of it have been decoded.

Boolean search modes, and ranking models that use $f_{t,i}$ values only, need no additional support beyond that which has been captured in Figure 3, and we explore two such options in Section 4. More complex querying can also be supported, provided that any additional necessary information is also made available. For example, ranked querying models in which a document-length normalization step is required must maintain a separate array of document lengths; we consider that to be not part of the core inverted index, and do not include it in our space measurements. Similarly, ranking modes that employ dynamic pruning heuristics need a per-term upper bound; that would require allocation of an additional 4-byte field in each head block, slightly increasing the size of the resultant index.

4 Experiments

We now describe a range of more detailed experiments that capture the balance of performance (see Figure 1) that is possible using our index structure.

4.1 Data and Hardware

Our experiments make use of three widely-accessible text collections:

- WSJ1 is the first half of the *Wall Street Journal* collection, consisting of newspaper text from 1987–1989, as used in several of the very early TREC activities.
- Robust04 is a collection of newswire documents from the 1980s and 1990s. It is made up of TREC Disks 4 and 5, excluding the *Congressional Record* from Disk 4.
- Wikipedia is a dump of the English Wikipedia corpus from April 2, 2022. Documents were extracted using the WikiExtractor[1] tool.

Each collection was first converted to a common *docstream* format prior indexing. A docstream represents documents as single lines of text, with the first element a document identifier, and the remainder of the line an ordered set of terms making up that document. A series of simple pre-processing steps was also applied as the docstreams were created:

[1] https://github.com/attardi/wikiextractor

	Words	Postings	Documents	Words/post.	Words/doc.	Size (MiB)
WSJ1	42,899,195	20,716,886	98,732	2.07	434.5	239.9
Robust04	278,511,531	121,987,739	528,155	2.28	527.3	1,551.3
Wikipedia	2,444,347,604	996,277,511	6,477,362	2.45	377.4	13,839.1

Table 5 Datasets used in experiments, after pre-processing. The final column refers to the docstreams prepared from the input text.

	Queries	Avg. Length	Avg. Conj. Match	Avg. Disj. Match
WSJ1	28,104	2.879	340	28,336
Robust04	28,104	2.879	2,001	155,554
Wikipedia	28,104	2.879	16,545	1,502,997

Table 6 Queries used in experiments, after the filtering step. Every query that was retained had at least one conjunctive match in each of the three collections.

long terms were broken after each group of 20 consecutive alphabetic characters; sequences of non-alphabetic characters were replaced with single spaces; and uppercase characters were folded to lowercase. No particular attention was applied to SGML or HTML markup elements, and the alphabetic components of those tags were retained in the docstreams. No stemming or stopping was performed either. Table 5 provides statistics of the resultant docstream collections that were then used in our experimentation. The largest of the three collections, Wikipedia, included more than two billion words, and gave rise to document-level indexes containing nearly a billion postings.

To generate test queries, all 60,000 queries from the TREC Million Query Track [1, 2, 15] were used as a starting point. Any queries that did not have a conjunctive match in each of the three test collections was then removed, to obtain a filtered log of around 28 thousand queries; the properties are summarized in Table 6. The last column gives the average number of documents that contained *any* of the terms in each query; the second-to-last the average number of documents that contained *every* query term.

Our experimental software was implemented in C++ and compiled with `gcc 7.5.0` with `-O3` optimization. All experiments were conducted on a Linux server with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM. Indexing experiments used an 894 GiB SATA SSD with transfer speeds of up to 5.6 GiB/s for all I/O. All experiments used a single processing core.

4.2 Experimental Objectives

We are now in a position to investigate the research goals that were listed in Section 3.1. In particular, we restrict our attention to the dynamic immediate-indexing part of a larger retrieval system that might also have some static index components (see Figure 2), and focus on an incoming stream of documents, so as to measure the indexing process and then the cost of querying against that dynamic index. Given that intention, the high-effectiveness compression regimes described in Section 2.2 cannot be employed, and nor can the query execution mechanisms described in Section 2.4. Both are relevant to the static shard index components, but not to the dynamic part. Focusing on the dynamic index component also means that we need to consider a suitable scale, and that is the purpose of the three selected datasets, each of which constitutes a plausible batch of updates in the context of a much larger retrieval system. For example, the Wikipedia collection contains around a billion postings, and an uncompressed index for it would require approximately 8 GiB of memory; it can be thought of as being 10%, or 1%, or even 0.1% of a "whole" retrieval system with the structure shown in Figure 2. Given that organization, and the fact that all aspects of our new mechanism scale linearly, the three test collections support experiments that are realistic in terms of the proposed operational environment.

Note also that while we primarily explore document retrieval using the standard document-at-a-time processing strategy, there is nothing in the new mechanism that precludes the use of term-at-a-time processing, since both options share the same document-sorted index requirement. On the other hand, score-at-a-time index processing is based on a different index organization, and would not be an option. Finally, note that the query time measurements reported below form an important complement to the "straight through" speeds already documented in Section 3.5, and help to locate the new mechanism in the trade-off space illustrated in Figure 1.

4.3 Baselines

Two previous implementations are used as reference systems. As a competitive baseline for both index size and query processing latency when a static index can be employed, and thus covering two of the corners of Figure 1, we use the C++ PISA system [44]. Two specific configurations were employed: one aimed at minimizing index space consumption via block-based interpolative coding [50] (denoted PISA-Interp); and one that provides a good balance between access time and space occupancy via the SIMD-BP128 bitpacking codec [36] (denoted PISA-BP128). The PISA software has been shown to be one of the fastest query processing systems available [41, 44, 45], and has also been independently compared to other open-source systems in terms of conjunctive Boolean query speed.[2] It thus provides an aspirational reference point for the new dynamic system in terms of both index size and query speed.

Then, in separate experiments that are presented in Section 5.2, we compare the new immediate-access indexing approach to the previous in-memory mechanism of Hawking and Billerbeck [32]. This second round of experimentation complements the aspirational PISA ones presented shortly, and allows a direct comparison with an alternative system which is also designed to operate in the upper half of the space shown in Figure 1.

4.4 Index Size

Table 7 gives an insight to the cost of the various components that make up a document-level index for WSJ1, for two typical values of the block size B. The dominant cost is the postings component of the full blocks, and they are where the heavy lifting gets done. Our design decision to not split postings across blocks has resulted in a small overhead cost in terms of unused bytes at the end of those blocks (around 0.3%–0.4% of total space) but also made both indexing and querying substantially more straightforward. Note also that more than two thirds of the terms never "escape" their head block to create a first tail block, and that the unused space at the end of head blocks is a non-trivial overhead, especially when $B = 64$.

[2] https://tantivy-search.github.io/bench/, accessed 30 May 2022.

Block and element	$B = 48$ bytes			$B = 64$ bytes		
	Blocks	Bytes	Percent	Blocks	Bytes	Percent
Head blocks	159,734			159,734		
– link pointers		1,277,872	3.1%		1,277,872	3.0%
– vocabulary		2,817,304	6.8%		2,817,304	6.6%
– postings		1,711,686	4.1%		2,362,446	5.6%
– trailing null bytes		1,860,370	4.5%		3,765,354	8.9%
Full blocks	632,105			449,644		
– link pointers		2,528,420	6.1%		1,798,576	4.2%
– postings		27,628,440	66.6%		26,851,357	63.2%
– trailing null bytes		184,180	0.4%		127,283	0.3%
Tail blocks	45,755			34,972		
– docnums		183,020	0.4%		139,888	0.3%
– postings		925,397	2.2%		955,899	2.2%
– unused bytes		1,087,823	2.6%		1,142,421	2.7%
Hash array		1,277,872	3.1%		1,277,872	3.0%
Total size	837,594	41,482,384	100.0%	644,350	42,516,272	100.0%

Table 7 Blocked index components (see Figure 3) and their relative contributions when indexing the WSJ1 collection, using two different block sizes B. All percentages are relative to the corresponding total index size in the last row, which includes the hash array, all vocabulary and global term statistics, all of the postings, and all overheads associated with the extensible lists including unused bytes at the end of allocated blocks. If a postings list contains only one block that block is counted as a head block, with the list not having a tail block.

Collection		$B = 40$	$B = 48$	$B = 56$	$B = 64$	$B = 72$	$B = 80$
WSJ1	doc-level	2.000	2.002	2.022	2.052	2.088	2.127
Robust04	doc-level	1.863	1.859	1.870	1.889	1.914	1.941
Wikipedia	doc-level	2.107	2.091	2.096	2.110	2.131	2.155

Table 8 Compression effectiveness for document-level indexes as block size B varies, in bytes per posting in all cases. Encoding parameter $F = 4$ was used throughout. All sizes include the hash table, terms and all other vocabulary information, plus the postings themselves. Block sizes less than 40 cannot be used. The best value in each row is shown in blue.

Indeed, there is more unused space in the head blocks than in the tail blocks, even though the latter are on average only half full, and even though the former also contain the term's vocabulary information.

Table 8 provides compression effectiveness results for all three collections and a range of typical values of B. These results are expressed as bytes per posting, and if required can be converted to megabyte sizes using the information provided in Table 5. The listed compression rates include all aspects of the index (accounted for in detail in Table 7 for two of the WSJ1 values in the first row), and provide an accurate assessment of the memory footprint

Collection		PISA-Interp	PISA-BP128
WSJ1	doc-level	1.162	1.514
Robust04	doc-level	1.075	1.515
Wikipedia	doc-level	1.318	1.800

Table 9 Compression effectiveness for document-level PISA indexes in bytes per posting, using two different compression regimes for postings, and including vocabulary and other files required for conjunctive querying. These results for static indexes can be compared against the corresponding dynamic index costs shown in Table 8.

that would be required if dynamic indexes had been constructed from the corresponding docstreams. The blue values show the best compression rates for each collection, with the variation away from those minima being relatively modest as B is varied. The Double-VByte compression technique (Algorithm 2) is an important contributor to the compact indexes that are created, along with the careful use (and re-use) of memory as detailed in Figure 3 and Algorithm 1.

Table 9 provides a reference point against which the results of Table 8 can be assessed. These come from the PISA system using the two different compression plug-ins [45] described above, and include all of the components needed to run a static in-memory retrieval system. As anticipated, both modalities attain better compression than is shown in Table 8, showing the benefit of knowing each posting list's characteristics and being able to tune compression parameters to match. The static PISA indexes also have no unused end-of-block space, nor any link pointers, both of which are inevitable in a dynamic index (see Table 7). Indeed, the relatively small ratio between the results in Table 9 and Table 8 is a positive outcome that indicates that the memory overhead incurred by dynamic indexing need not be excessive.

Combining the information in Tables 5 and 8 indicates that a dynamic index of 256 GiB (the upper limit possible with $B = 64$ and $h = 4$) will be able to support more than 100 billion postings, corresponding to approximately 200 billion words in typical-length documents, and hence to more than 1 TiB of plain text.

4.5 Indexing Throughput

Figure 4 shows how fast indexing is with our proposed mechanism, reporting indexing times measured in units of microseconds per document. Each collection was processed twice: a first time with all of the calls to *add_posting*() (see Algorithm 1) returning immediately, without doing any work, shown as "Count only"; and then a second time with the *add_posting*() calls all fully functional. The difference in the two times is then cost of index construction, with the balance of the time being required to read the input, parse it into tokens, and count the term frequencies within each document – processing that is required by all indexing implementations.

The *add_posting*() calls take only a small fraction of the indexing time. For example, on Wikipedia the Count only time is 420.7 seconds, and Count+Index was 574.5 seconds, or over a gigabyte per minute. As further reference points, Lin et al. [40] reported the production-ready Lucene system to achieve up to 5.1 GiB per minute when indexing the ClueWeb12B collection using 48 processing threads, including tokenization; and Hawking and Billerbeck [32] report a "List Building" time of around 320 seconds for a collection of approximately one billion

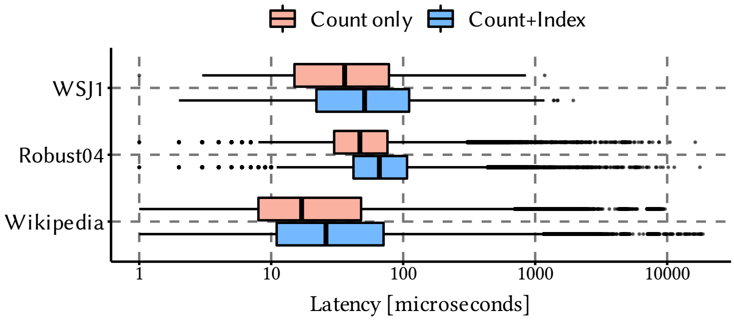

Figure 4 Per-document insertion time in microseconds, with "Count only" the time taken to tokenize and count the terms in each document without inserting them into the index, as a lower-bound on indexing cost; and with "Count+Index" then including the cost of inserting the postings as well.

word-level postings (their Table 7, row "FibonacciB, Large Pages"). The latter value is validated in Section 5.2.

4.6 Query Throughput

We investigate response latency using two separate query modes: Boolean conjunctions in a document-at-a-time manner (see Culpepper and Moffat [20, Algorithms 1 and 3]), returning a list of all matching document identifiers; and via a document-at-a-time top-k disjunctive retrieval strategy. In the latter mode the top-k documents "seen so far" are tracked in a min-heap structure, with each document that contains any of the query terms scored and compared against the smallest element in the heap. In the PISA baseline system scores were computed according to the common BM25 model [53] (the precise formulation used can be found in the PISA overview [44]). In our system we employed a simple TF×IDF ranking model in which the weight w of term t in document d is computed as

$$w_{t,d} = \log(1 + f_{t,d}) \times \log(1 + N/f_t)\,,$$

where $f_{t,d}$ is the frequency of t in d, where f_t is the number of documents that contain t, and where N is the number of documents in the collection; and created answer sets containing the top $k = 10$ answers for each query. Note that our interest here is on retrieval speed, and that we are not claiming that the similarity formulation employed is competitive in terms of retrieval effectiveness. Note that this experiment is not designed to compare like-with-like. As already noted, the PISA comparator system makes use of pre-computed and pre-optimized static indexes; in addition, there are also differences in the two similarity formulations that might affect their relative speed. In particular, when the BM25 computation embedded in PISA was replaced by a TF×IDF version, query processing became approximately 25% slower. When that relativity is taken into account, this experiment gives helpful guidance in regard to the difference in cost between querying using an optimized static index, and querying using our dynamic index, which is designed for versatility. In a full system both types of querying might be required, as is shown in Figure 2.

Figure 5 shows per-query query latency distributions for two versions of the PISA reference system, in both cases making use of static indexes, static querying modes, and the faster BM25 computation, comparing them to our proposed dynamic index arrangement. The six panes represent three collections and two querying modes. While both PISA-based systems

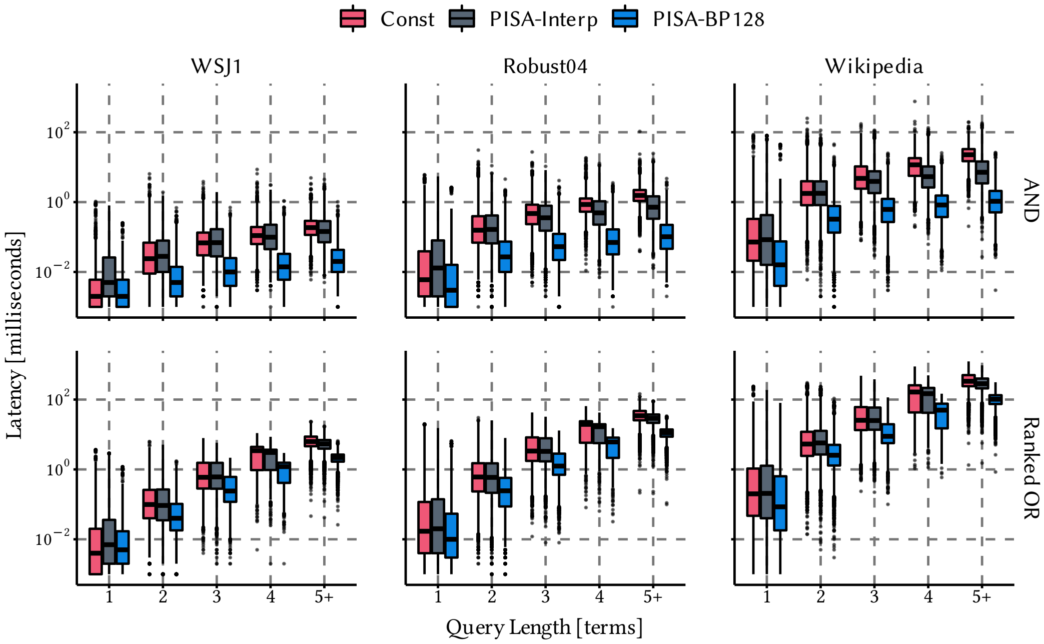

Figure 5 Query processing time distribution in milliseconds per query, using the filtered MQT query log (Table 6) and plotted as a function of the number of terms in each query. Each graph compares our dynamic index with two PISA-based systems, with the six graph panes covering three collections (Table 5) and two querying modes (conjunctive Boolean, and top-10 disjunctive).

are (as expected) faster than ours, nor is our system a laggard. For example, conjunctive queries on the Robust04 collection can be evaluated in around 550 microseconds on average for queries of up to four terms, and the more expensive ranked disjunctive queries of the same length can be resolved in around five milliseconds on average.

5 Extensions

In this section we consider several extensions to the scheme described in Section 3 and measured in Section 4. In particular, we consider word-level indexing, showing that Double-VByte is again a useful technique, albeit with a twist (Section 5.1); we consider variable block sizes, introducing a new extensible list approach that allows an asymptotic reduction in the overhead cost ratio (Sections 5.3 and 5.4); and we describe a simple block rearrangement mechanism that accelerates query processing speed and involves only a minor interruption to the document ingestion process (Section 5.5).

5.1 Word-Level Indexing

The structure and approach described in Section 3 can be used to construct word-level indexes with only small modifications required. Postings $\langle d_{t,i}, w_{t,i,j}\rangle$ (see Table 1) in which $d_{t,i}$ is the ordinal number of the ith document containing term t, and $w_{t,i,j}$ is the ordinal word number within d of the jth instance of t, are coded as two values. The first value is a d-gap relative to the immediately preceding posting, but then with one added, to ensure that the resultant $g_{t,i}$ value is strictly greater than zero (Section 2.2). The second element in each

New size (Double-VByte, bytes)	Original posting size (VByte, bytes)			
	2	3	4	5
	56.17%	36.91%	6.57%	0.36%
1	30.22%	–	–	–
2	18.97%	14.30%	–	–
3	6.98%	21.11%	0.05%	–
4	–	1.50%	6.43%	0.00%
5	–	–	0.08%	0.36%

Table 10 Percentages of $\langle d_{t,i}, w_{t,i,j} \rangle$ postings of each given length in the WSJ1 collection when represented as separate VByte codes for the $d_{t,i}$ and the $w_{t,i,j}$ components, and the distribution of altered sizes when jointly coded via Algorithm 2, using $F = 3$.

Collection		$B = 40$	$B = 48$	$B = 56$	$B = 64$	$B = 72$	$B = 80$
WSJ1	word-level	2.614	2.571	2.550	2.542	2.542	2.548
Robust04	word-level	2.414	2.370	2.346	2.334	2.328	2.328
Wikipedia	word-level	2.499	2.448	2.420	2.404	2.395	2.392

Table 11 Compression effectiveness for $\langle d_{t,i}, w_{t,i,j} \rangle$-format word-level indexes as a function of block size B, in bytes per posting. Encoding parameter $F = 3$ was used throughout. All sizes include the hash table, terms and all other vocabulary information, plus the postings themselves. The best value in each row is shown in blue.

posting is a corresponding w-gap that represents the ordinal word number, if this is the first occurrence of t in $d_{t,i}$, or represents the interval since the most recent previous instance of t, if this is $d_{t,i}$'s second or subsequent occurrence of t. In this arrangement the term occurrence count $f_{t,i}$ becomes a derived quantity, calculable by scanning the postings list.

In a word-level index every term in the input generates a posting. That means that there are many d-gaps of just one, arising when the term appears multiple times in the same document; and also a smaller but still significant fraction for which the d-gap is two, corresponding to t appearing in consecutive documents. On the other hand, the w-gaps tend to be larger, since it is relatively rare for a word to closely follow itself in a document (with the exception of some song lyrics, “she loves you, yeah, yeah, yeah”).

The Double-VByte approach can again be used to create compound codes, but should be applied with the arguments swapped (the “twist” mentioned above), so as to exploit that reversal of the two relative frequency distributions. In our word-level indexes, an adjusted $\langle g, w \rangle$ combination is encoded using $double\_vbyte\_encode(w, g)$ (and using $F = 3$), whereas in the document-level indexes considered in Sections 3 and 4 each $\langle g, f \rangle$ combination was represented via a call to $double\_vbyte\_encode(g, f)$. Table 10 demonstrates that this minor adaptation again leads to substantial compression savings, with around 45% of the postings becoming one byte shorter and less than 9% becoming one byte longer.

Word-level indexes contain more information and are thus more expensive to store than document-level ones. Table 11 gives a full range of results, with the listed bytes/posting rates again including the hash array, the postings overhead, and all of the extensible list overheads, and hence comparable with the values reported in Table 8. For word-based indexes employing

Approach	Time (seconds)	Space (MiB)
Hawking and Billerbeck [32]	329	18,114
Ours	545	5,603

Table 12 Immediate-access indexing techniques, comparing our implementation with that of Hawking and Billerbeck [32], building a word-level index for the Wikipedia collection.

the Const strategy the best compression effectiveness is achieved with somewhat larger block sizes. Note that in Table 8 the compression rates are also given in bytes per posting, but that there are more postings in a word-level index than in a document-level index. Table 5 gives exact ratios connecting these various quantities.

The results in Table 11 compare very favorably with the 6–7 bytes per posting (that is, per input word) achieved by the best method of Hawking and Billerbeck [32, Table 6, method FibonacciB] for the same mode of word-level indexing. They also compare well against the rates achieved by Büttcher and Clarke [12] for their schema independent word-level index (see the last row of Table 1 in Section 2.1). In particular, Büttcher and Clarke report (in their Table 4) space consumptions that equate to approximately 1.8 bytes per posting (method $\text{Explimit}_{16,1.1,512}$), but with that rate not including the cost of the additional document-to-word mapping that is required, nor the cost of the term vocabulary structure. The schema-independent scheme is also unlikely to resolve queries as quickly as our approach, because of the need to consult the document mapping for each posting. It thus represents a further option in the trade-off space illustrated by Figure 1.

5.2 Compared to Previous Immediate-Access Indexing Approaches

Table 12 compares our new mechanism with the code of Hawking and Billerbeck [32], building a $\langle d_{t,i}, w_{t,i,j}\rangle$-style word-level inverted index for the Wikipedia collection in each case. The Hawking and Billerbeck code executes more quickly than does ours, but our software generates an index that requires only 30% of the space. That is, the careful attention we have given to minimizing the memory footprint means that our code can process more than three times as many documents in a given amount of main memory before needing to transfer index data to secondary storage.

5.3 Variable Block Indexing

The mechanism described in Section 3 uses uniform-sized blocks, each of B bytes, which is the Const_B approach of Büttcher and Clarke [12] and Hawking and Billerbeck [32] (noting that in some descriptions B is counted in postings rather than in bytes; here we primarily make use of quantities measured in bytes). The Const approach has a number of advantages, primarily the simplicity of addressing, but has the disadvantage of being non-adaptive to term frequencies – rare terms are treated exactly the same as common ones. In particular, if h bytes of linking (pointer) information are required in each block, leaving $B-h$ bytes available for the payload bytes storing vocabulary information and postings, then constant-sized blocks result in a minimum *asymptotic overhead ratio* of $h/(B-h)$. For example, if $B = 64$ and $h = 4$, the minimum asymptotic overhead ratio of non-payload bytes to payload bytes is 6.67%. There will also be up to $B-1$ unused bytes in the last block in the chain, which pushes the average overhead ratio higher.

A range of non-constant approaches were also explored by Büttcher and Clarke [12] and Hawking and Billerbeck [32], in particular, a range of *exponential growth* schemes, denoted $\mathsf{Expon}_{B,k}$. In the $\mathsf{Expon}_{B,k}$ scheme B is the first block size, and $k \geq 1$ is a growth parameter. The first block in each chain contains B bytes in total, including h bytes of link overhead; the second contains a total of $\lceil h + (B-h)k \rceil$ bytes; the third $\lceil h + (B-h)k^2 \rceil$ bytes; and so on, forming a geometric sequence of ever-increasing payload capacities. While this approach reduces the number of blocks in each chain from being linear in the total payload volume to being logarithmic in the payload volume, and thus similarly reduces the number of pointers required, it doesn't reduce the average asymptotic overhead ratio. In particular, as the blocks become larger, the average number of unused bytes in the tail block also grows exponentially. Asymptotically, if the block-on-block payload growth rate is by a factor of exactly k, then the tail block contains a fraction of the total payload capacity that can be calculated as

$$\lim_{z\to\infty} \frac{B'k^z}{B' + B'k + B'k^2 + \cdots + B'k^z} = \lim_{z\to\infty} \frac{k^z}{(k^{z+1}-1)/(k-1)} \approx \frac{k-1}{k}\,, \tag{1}$$

where $B' = B - h$ is the payload capacity of the first block, and where z is the index of the current tail block. That is, in a postings list allocated with a current total capacity of n payload bytes, approximately $n(k-1)$ of those payload bytes are in the tail block; moreover, when averaged across all postings lists and as k approaches 1 from above, approximately half of the index's tail block bytes will be unused at any given time. For $k = 1.1$, the growth rate favored by Büttcher and Clarke [12], that implies an average asymptotic overhead ratio in the vicinity of 5%, not including the approximately $\log_k(n/B)$ link pointers, each consuming h bytes.

That is, growing the blocks exponentially using any fixed radix k incurs an overhead ratio that is again (in an amortized sense) linear in the volume of stored payloads. In both the Const and Expon cases the exact asymptotic rate is determined by the parameter choice (B and k, assuming h is fixed), and hence can be controlled to a certain extent. But setting the parameters to minimize the asymptotic rate also has the perverse effect of increasing the actual measured cost in typical non-asymptotic (that is, practical) situations. Care – and a degree of empirical exploration to establish exact overhead rates as percentages of the payloads stored [12, 32] – is required if space is to be minimized.

Schemes that exponentially grow until some upper limit is reached and then revert to constant block sizes [12], and schemes based on Fibonacci numbers or that have a number of same-size repeat blocks at each block size [32] all share this same asymptotic behavior – the expected overhead converges to a fixed linear fraction of the total payload volume.

5.4 Extensible Lists Revisited

To reduce the asymptotic overhead ratio a different approach is required. To set the scene, suppose fixed blocks of B bytes each (including the h bytes required for the block's link pointer) will be employed, and that a total payload of n bytes is to be accommodated, but with n unknown. Then the number of blocks required is $z = \lceil n/(B-h) \rceil$; and the total overhead cost will be given by $W = Bz - n$, covering both the links and the unused payload slots in the tail block. If we assume that n is large enough that $z \approx 0.5 + n/(B-h)$ (because we don't actually know the value of n, but can assume that when taken modulo $B - h$ it falls mid-way through the range from 0 to $B - h - 1$) then the wastage W is

$$W = B \cdot \left(0.5 + \frac{n}{B-h}\right) - n = 0.5 \cdot B + n \cdot \frac{B}{B-h} - n\,.$$

Making the further assumption that $h \ll B$, so that $B/(B-h) \approx 1 + h/B$, we then get

$$W = 0.5 \cdot B + \frac{hn}{B} + n - n = 0.5 \cdot B + \frac{hn}{B}\,.$$

Taking h and n to be fixed, and choosing B to minimize W then leads to:

$$\frac{\partial W}{\partial B} = 0.5 - hn/B^2$$

which is equal to zero when

$$0.5 = hn/B^2$$

that is, when

$$B = \sqrt{2hn}\,. \tag{2}$$

As an example of what this means, consider the case when $h = 4$ bytes and a list has $n = 20{,}000$ payload bytes. Then a block size of $B = 400$ bytes is computed from Equation 2, and the resultant postings list arrangement would have $z = \lceil 20{,}000/396 \rceil = 51$ blocks, a total of 20,196 available payload bytes, a total of 204 bytes of link pointers, and a further 196 bytes of unused space in the tail block, for a total overhead of 400 bytes. Note how the overhead ratio is minimized when the overhead consists of approximately equal volumes of link pointer bytes and unused tail block bytes.

Of course, the problem is that we don't know the final value of n. Indeed, there is no final value, n is perpetually subject to upward revision as the index is extended. But n does have a *current* value, and we know exactly what it is – it is the sum of the payloads stored in the current chain of blocks. In particular, suppose that the zth block of each postings list is of size B_z and carries a payload of $p_z = B_z - h$ bytes. Suppose further that the first z blocks of the posting list are full and hence that $n = \sum_{i=1}^{z} p_i$, and that we must determine a block size for the $z+1$th block. For the Const_B approach [12, 32] the situation is simple:

$$B_{z+1} = B_z = B\,, \tag{3}$$

as already noted in Section 2.5. The relationship captured in Equation 1 means that the $\mathsf{Expon}_{B,k}$ method with growth parameter k is also easy to now state:

$$B_{z+1} = h + (k-1) \cdot \sum_{i=1}^{z} p_i = h + (k-1) \cdot n\,. \tag{4}$$

In practice we may wish to use block sizes that are integer multiples of the base unit B, and in this case we take $B_1 = B$, and then B-align each subsequent block B_z, taking care that the minimum block size allocated is B:

$$B_{z+1} = B \cdot \left\lceil \frac{h + (k-1) \cdot n}{B} \right\rceil\,. \tag{5}$$

For example, when $B = 16$, $h = 4$, and $k = 1.5$, the sequence of block sizes will be $B_z = \langle 16, 16, 16, 32, 48, 64, 96, 144, 208, \cdots \rangle$ bytes.

Given the context established by the Const and Expon approaches and their definitions via Equations 3 and 5, we now define a new growth mechanism for extensible lists that exploits the relationship captured in Equation 2. To understand the basis of this new Triangle strategy, consider Figure 6, which shows a simple example that commences with an initial

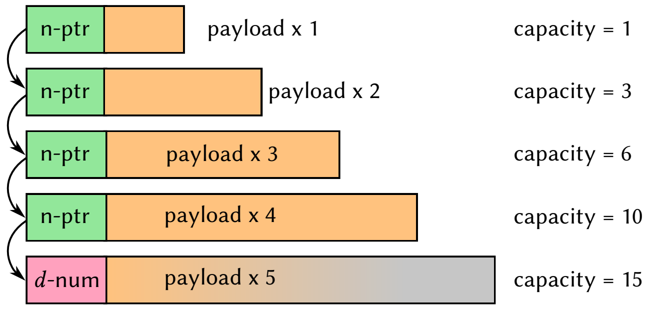

Figure 6 The basic Triangle method for extensible lists, starting with a single payload, and with each link pointer requiring the same space as a single payload (that is, $B = 2$ and $h = 1$). The zth block in the chain contains z payloads, and the total payload capacity in the first z blocks is $n = z(z + 1)/2$.

block of size $B = 2$ and requires $h = 1$ cells for each link pointer. In the figure the zth block allocated always contains space for z payload slots, plus a link pointer. Because $\sum_{i=1}^{z} i = z(z + 1)/2 \approx z^2/2$, once B_z is allocated the structure contains approximately $z^2/2$ payload slots, and z links. Moreover, the worst overhead ratios occur straight after each zth block is added, when there is just one payload slot within it that has been consumed. At that moment the structure contains z link pointers and $z(z + 1)/2$ payload slots, of which $n = (z - 1)z/2 + 1$ are in use and $z - 1$ are vacant. The overhead ratio at this point in the growth cycle is thus

$$\frac{2z - 1}{n} = \frac{2z - 1}{z^2/2 - z/2 + 1} \approx \frac{4}{z} \approx \frac{2\sqrt{2}}{\sqrt{n}} .$$

This is asymptotically superior to both the Const and Expon strategies, which have overhead ratios that are constant. Note the close connection between this example and the conclusion captured in Equation 2. In Figure 6 the relationship $n = z(z + 1)/2$ means that for a given value n, the largest block will be of size $B_z \approx \sqrt{2n}$, and hence that the next block will be one larger: $B_{z+1} \approx 1 + \sqrt{2n}$, matching Equation 2 in the $h = 1$ case.

More generally, the Triangle$_B$ method (named as a consequence of the visualization shown in Figure 6 for the simplest $B = 2$ and $h = 1$ case) is defined via:

$$B_{z+1} = B \cdot \left\lceil \frac{h + \sqrt{2hn}}{B} \right\rceil \tag{6}$$

in which n is again the total payload volume at the moment the expansion is required. With this formulation block sizes grow as the square root of what has already been accommodated in the list, a more sedate pattern than the rapid Expon acceleration. For example, when $B = 16$ and $h = 4$, Equation 6 leads to the sequence of block sizes $B_z = \langle 16, 16, 32, 32, 32, 48, 48, 48, 48, \cdots \rangle$, and hence to the sequence of payload capacities $p_z = \langle 12, 12, 28, 28, 28, 44, 44, 44, 44, \cdots \rangle$.

Figure 7 show the pattern of non-payload storage required, as a function of the total volume of payloads, here with first blocks of size $B = 16$, and with $h = 1$ cells required by each link pointer and thus unavailable as payload (a scenario that corresponds to $B = 64$ and $h = 4$ when the unit of measurement is bytes). The three sawtooth lines show the exact overhead count associated with input payload volume for the three different growth methods, with the sawtooth jumping to a new high point each time an additional block is allocated,

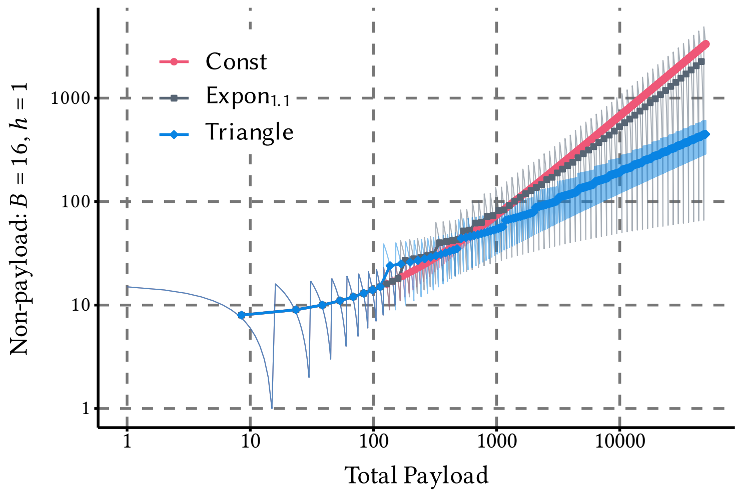

Figure 7 Volume of non-payload storage required, plotted as a function of the total volume of payload data, where the first block contains $B = 16$ units of payload, and where $h = 1$ element within each block is required for the block link information. The $\mathsf{Expon}_{16,1.1}$ and $\mathsf{Triangle}_{16}$ approaches are defined by Equations 5 and 6 respectively, with block sizes B_z that are always integer multiples of B.

Method	Document-level		Word-level	
	$B = 48$	$B = 64$	$B = 48$	$B = 64$
Const_B	2.091	2.110	2.448	2.404
$\mathsf{Expon}_{B,1.1}$	1.996	2.056	2.263	2.282
$\mathsf{Triangle}_B$	1.952	2.013	2.226	2.246

Table 13 Total index cost (bytes per posting, comparable with Tables 8 and 11) for indexes for Wikipedia using three different growth methods for extensible lists and two different block sizes, both in conjunction with $h = 4$. In the $\mathsf{Expon}_{B,k}$ and $\mathsf{Triangle}_B$ methods the vocabulary in each head block (see Figure 3) requires two extra bytes compared to Const_B, to handle the variable block sizing. The best value in each column is shown in blue.

and then decreasing through a cycle as that block gets filled. The three smooth lines though the marked dots represent the average overhead within each of the sawtooth growth cycles, and thus indicate the long-term amortized overheads of the three methods. The red (Const) and grey ($\mathsf{Expon}_{16,1.1}$) lines become parallel to each other in this log-log graph as the payload volume increases, showing that they have the same asymptotic growth rate.

On the other hand, the overhead ratio for the new Triangle strategy is a sub-constant fraction of the number of payload words, a relationship confirmed by the reduced gradient of the corresponding blue line in the log-log graph. Hence, while Triangle is slightly less compact than both Const and Expon over small sections of the range of values n, it always becomes more efficient than both Const and Expon on long lists. In Figure 7 the last full Triangle growth cycle covers lists containing between $n = 49{,}681$ and $n = 49{,}999$ payload items inclusive, an average of $n = 49{,}840.5$; and gives rise to an average overhead ratio of just 0.90% through that growth cycle.

The critical question then is what happens in practice for a typical mix of postings lists lengths. If most lists are long, or if some lists are very long, Triangle is likely to lead to the most compact index. On the other hand, if there are many lists that fall into the middle band of lengths, then one of the other two methods might be the most compact in a practical (rather than asymptotic) sense. Another factor to be accounted for is increased complexity – each head block (see Figure 3) needs an additional field inserted, to track the current block size B_z (or rather, to track z); and the field nx must also be extended, since 256 bytes is no longer a plausible upper bound on block size. In our implementation these small changes add two bytes to every head node, with block sizes capped at 2^{16} bytes and nx becoming a two-byte integer, and with z a one-byte integer and capped at 256.

Table 13 shows compression effectiveness, returning to $h = 4$, with $B = 48$ and $B = 64$, with n counted in bytes, with two extra vocabulary bytes for the Expon and Triangle methods, and with all other aspects as described in Section 3. The Triangle approach secures small additional savings in all cases compared to both the Const strategy and the Expon approach.

However, there is a drawback to both the Expon and Triangle approaches, a result of their ever-lengthening postings blocks. In conjunctive querying modes (and also in ranked querying modes when implemented using a dynamic pruning mechanism such as MaxScore or WAND [8, 22, 41, 63]) the *seek_GEQ*() operation plays an important role, bypassing blocks that are not required. But the likelihood of bypassing any particular block decreases as the blocks become longer, because each block contains more postings. That effect is then compounded, because more sequential decoding effort is also required to reach any given document number within a long block if it does get decoded. In combination those two effects mean that while plain disjunctive queries operate at similar speeds in all of Const, Expon, and Triangle approaches, conjunctive evaluation is fastest with the Const extensible list structure. This tension provides yet another example of the space-speed-indexing tradeoffs that were illustrated in Figure 1.

5.5 Periodic Collation

As described in Section 3, in the Const approach the postings list for each term t consists of a chain of blocks, all of some fixed length B, each separated from the other blocks associated with t by blocks allocated to other terms. Access to each new block when following t's postings chain might thus result in a cache-miss (or equivalently, a pre-fetch failure), since the intervals between t's blocks are likely to be highly variable. This, in turn, means that the indexed sequential access associated with the *seek_GEQ*() operations is relatively costly, since only a few bytes of memory (a b-gap and then the *n_ptr*) are accessed from within each block, with another long memory jump often immediately following.

To ameliorate that cost, the final enhancement we propose is that of *collating* the index blocks. This is a very simple operation – somewhat akin to the "list traversal" phase discussed by Hawking and Billerbeck [32] – that can be undertaken periodically, perhaps when indexing/querying load is low. To get started, a copy of the hash array is made, with ingest operations temporarily suspended and querying operations carried out as usual via the old hash array. Each non-empty element in that copied hash array, corresponding to some term t, is then visited. First, t's head block is written from $\mathcal{I}$ to a sequential disk file, and then each of t's other blocks is written, through to and including the tail block; all the while replacing the *n_ptr* and *t_ptr* fields with revised values determined by a counter of blocks written to disk. No other alterations are made to any of the blocks, and they retain all of their remaining components completely unchanged. As terms are processed, the entries in the copied hash array are updated with the new offsets of the corresponding head blocks

Collection	Conjunction		Top-10 disjunction	
	Mean	P_{95}	Mean	P_{95}
Const, interleaved	8.53	32.03	90.29	455.96
Const, collated	4.08	13.94	83.87	426.36
Triangle, interleaved	31.49	123.67	83.78	424.41
Triangle, collated	31.07	122.17	82.63	419.93

Table 14 Query times for the Wikipedia collection for two different querying (Section 4.6), with indexes formed via the Const and Triangle strategies with $B = 64$ and $h = 4$. Each pair of rows shows the query time with postings blocks in "arrival order" in the index array $\mathcal{I}$, and then after the collation process. All times are in milliseconds per query, and show the mean across the query set (Table 6) and the 95 th percentile query execution time across the query set.

in the reordered on-disk version of the index. This relatively lightweight process results in a "collated" version of $\mathcal{I}$ on the disk. A brief pause in query processing is then required while a single binary read operation brings the permuted index back into memory, overwriting the exact same space within $\mathcal{I}$. At the same time the new hash array replaces the old one; once that is done, document ingestion and querying can be resumed.

The post-collation index still has the structure shown in Figure 3, and it is only the interleaving of the blocks within $\mathcal{I}$ that is affected by the collation process – the index remains both queryable and extensible. However in the collated index all of each term's blocks are stored contiguously, a change that under some circumstances notably improves query processing times. Table 14 supports that claim. The query set used in Section 4.6 was executed against four indexes for the Wikipedia dataset. The first index is as described in Section 3 and already measured in Section 4, and is constructed using the Const strategy. The second is a collated version of that, still with all blocks the same length, occupying $B = 64$ bytes each. The third of the three indexes is an uncollated index constructed using the Triangle strategy, which saves a small amount of space compared to Const; and the fourth index is a collated version of the third, still using the Triangle pattern of block lengths, but now with each term's postings in a continuous section of $\mathcal{I}$.

As can be seen, when the index in constructed using the Const approach, conjunctive querying time is reduced by a factor of up to two in the collated index compared to the "postings arrival order" original interleaved index, with tail latency (the two P_{95} columns) also notably improved. On this Wikipedia collection, collation allows our system to outperform the compact PISA-Interp system which has a mean latency of 5.74 milliseconds per query (compared to the 4.08 milliseconds shown in Table 14). Detailed profiling experiments showed that the majority of this speedup was due to improved cache behavior, with around 66% fewer cache misses after collation. There are also small gains possible in connection with disjunctive queries.

On the other hand, when the index is constructed using the Triangle strategy, collation has little effect on execution times, since the long postings lists are already represented as a small number of long blocks. Moreover, the long blocks of the Triangle approach actively hinder conjunctive query evaluation, an effect noted earlier in Section 5.4. With long postings blocks for common terms, skip-search using *seek_GEQ*() operators is much less effective than it is with the Const strategy and its small blocks, suggesting that the collated Const mechanism provides the best overall balance of features.

The collation process also has an associated cost. On the Wikipedia collection, writing the complete index to SSD in a single operation (that is, without collation) requires around 0.9 seconds; the sequence of random accesses into $\mathcal{I}$ and the large number of small $B = 64$-byte writes during collation extends that time to 6.7 seconds. During that whole collation period ingestion must be stalled, meaning that queries that arrive during that period will not be truly immediate-access, with documents that arrived during the last approximately 7.5 seconds not retrievable. If that risk is acceptable, then collation allows subsequent queries to be resolved more quickly, thus providing yet another trade-off option in the space shown schematically in Figure 1. It may also be possible to handle tail blocks in a more strategic manner and allow concurrent ingestion and collation, an option that we plan to explore in future work.

5.6 Managing Large Arrays

A key element of our proposal is the use of a large array of B-byte blocks. If sufficient memory can be allocated to that array as a monolithic segment, then all is well, and an immediate-index can be constructed within that slab of memory. On the other hand, if multiple indexes must be constructed at the same time – for example, if the arriving document stream contains multiple languages, and each language is to be indexed independently – it may be desirable for the required arrays of blocks to somehow co-exist within the same total amount of memory space, with each of them growing as required until the overall envelope of space has been exhausted. In separate work we consider how to achieve that goal [49].

6 Discussion and Conclusions

We have developed new insights into how dynamic retrieval systems can be structured and organized. Furthermore, as specific and concrete innovations, we have described a byte-packing mechanism that greatly improves the compression effectiveness of the previous VByte approach, and have also developed a new extensible list technique that is asymptotically more efficient than any of the mechanisms that have been considered previously. The second improvement reduces the overhead storage space to retain n items from being linear in n (that is, $\Theta(n)$) to being sub-linear in n (to be precise, $\Theta(n^{1/2}) \in o(n)$). Both of these enhancements represent significant developments in the way that dynamic indexes are structured and stored.

As a broader assessment of what we have developed in this project, if information retrieval was an athletics contest, then our structure wouldn't be the fastest sprinter (because for sheer query speed, the application of document reordering to a static document pool, with variable length blocks, and MaxScore- or WAND-based dynamic pruning would result in faster query processing); nor would it be the highest jumper (because for minimal memory footprint, interpolative coding and/or localized parameterized codes are better); and it might not win the shot put either (because for sheer indexing speed, an offline sort-based approach might be preferable).

But those "better" options – occupying the corners in the triangle in Figure 1 – do not offer dynamic indexing and immediate-access querying, and hence cannot be used for every component of a system of the type shown in Figure 2. Given the desire to provide real-time support for document ingestion and online search, the event (to continue the athletics metaphor) we are competing in is the heptathlon, in which an all-round performer must be competent across multiple dimensions, so as to achieve a superior overall balance of performance. In describing our fixed-block inverted file structure and the immediate-access querying that it supports, and documenting its performance relative to a carefully-engineered offline reference system, we have explored the interior of the triangle in Figure 1, and have presented an Olympic-level heptathlete that will be of considerable interest to practitioners and researchers alike.

6.1 Limitations

Our scheme does, of course, have limitations. We have not considered document deletions nor updates, and at present our software only handles "append document" and "keyword search" operations. Deletions are a complex challenge in all inverted file-based retrieval systems, because of the need to withdraw postings from the middle of lists, and then adjust neighboring d-gaps; and the immediate-index we have described is no different in that regard. We have also only considered query processing in a "proof of concept" manner in the experiments described in Section 4. For example, mechanisms of the type summarized in Section 2.4 that make use of stored block upper bounds might allow faster similarity-based top-k query processing, with only small growth of the index being needed.

Nor have we considered the integration of the immediate-access part of the index with the static shards. As with all distributed systems, load balancing is important, and in focusing entirely on the immediate-access index, we have deferred consideration of how the shards shown in Figure 2 will interact with each other. In addition, we have not considered concurrency control and the extent to which low-level operations require locking, and have assumed in this discussion that all postings associated with each ingested document are processed into the index before the next query operation is permitted. Finer-grained analysis of querying and document insertion, and of in-parallel insertion of multiple documents, are also necessary in a production system.

6.2 Future Work

Each of those limitations is also an opportunity for future work. Our immediate next goal will be to consider how to integrate responsive querying modes that – with as little additional data being stored as is possible – allow more precise disjunctive skipping to take place, so that we can reduce the time taken to carry out ranked bag-of-words querying using similarity scoring models such as BM25.

Software

Public software that implements our method is available from `https://github.com/JMMackenzie/immediate-access`

Acknowledgements

We thank the referees for their detailed and helpful input. This work was in part supported by the Australian Research Council (project DP200103136).

References

1 J. Allan, B. Carterette, J. A. Aslam, V. Pavlu, B. Dachev, and E. Kanoulas. Million query track 2007 overview. In *Proc. Text Retrieval Conf. (TREC)*, 2007.

2 J. Allan, J. A. Aslam, B. Carterette, V. Pavlu, and E. Kanoulas. Million query track 2008 overview. In *Proc. Text Retrieval Conf. (TREC)*, 2008.

3 D. Arroyuelo, M. Oyarzún, S. González, and V. Sepulveda. Hybrid compression of inverted lists for reordered document collections. *Information Processing & Management*, 54(6):1308–1324, 2018.

4 N. Asadi, J. Lin, and M. Busch. Dynamic memory allocation policies for postings in real-time Twitter search. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1186–1194, 2013.

5 X. Bai and B. B. Cambazoglu. Impact of response latency on sponsored search. *Information Processing & Management*, 56(1):110–129, 2019.

6 X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Trans. on Information Systems*, 36(2):1–42, 2017.

7 N. R. Brisaboa, A. Farina, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 122–136. Springer, 2003.

8 A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 426–434, 2003.

9 E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. Conf. on Very Large Databases (VLDB)*, pages 192–202, 1994.

10 M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 1360–1369, 2012.

11 S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 317–318, 2005.

12 S. Büttcher and C. L. A. Clarke. Memory management strategies for single-pass index construction in text retrieval systems. Technical Report CS-2005-32, University of Waterloo, 2005. URL `http://www.stefan.buettcher.org/papers/wumpus-tr-2005-02.pdf`.

13 S. Büttcher and C. L. A. Clarke. Hybrid index maintenance for contiguous inverted lists. *Information Retrieval*, 11(3):175–207, 2008.

14 S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and evaluating search engines.* MIT Press, Cambridge, MA, 2010.

15 B. Carterette, V. Pavlu, H. Fang, and E. Kanoulas. Million query track 2009 overview. In *Proc. Text Retrieval Conf. (TREC)*, 2009.

16 M. Chang and C.-K. Poon. Efficient phrase querying with common phrase index. *Information Processing & Management*, 44(2):756–769, 2008.

17 G. Chowdhury. An agenda for green information retrieval research. *Information Processing & Management*, 48(6):1067–1077, 2012.

18 C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Dynamic inverted indexes for a distributed full-text retrieval system. Technical Report CS-94-40, University of Waterloo, 1994. URL `https://www.researchgate.net/publication/2699270_Fast_Inverted_Indexes_with_On-Line_Update`.

19 J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 1–12, 2005.

20 J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. on Information Systems*, 29(1):1.1–1.25, 2010.

21 D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 405–411, 1990.

22 C. M. Daoud, E. S. de Moura, A. L. Carvalho, A. S. da Silva, D. Fernandes, and C. Rossi. Fast top-k preserving query processing using two-tier indexes. *Information Processing & Management*, 52(5):855–872, 2016.

23 C. M. Daoud, E. S. de Moura, D. Fernandes, A. S. da Silva, C. Rossi, and A. Carvalho. Waves: A fast multi-tier top-k query processing algorithm. *Information Retrieval*, 20(3): 292–316, 2017.

24 E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems*, 18(2):113–139, 2000.

25 J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proc. Conf. on Web Search and Data Mining (WSDM)*, pages 1–1, 2009.

26 L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1535–1544, 2016.

27 S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002, 2011.

28 P. Eades, A. Wirth, and J. Zobel. Immediate text search on streams using apoptosic indexes. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 157–169, 2022.

29 C. Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *Proc. Conf. on Very Large Databases (VLDB)*, pages 363–374, 1992.

30 E. A. Fox and W. C. Lee. FAST-INV: A fast algorithm for building large inverted files. Technical Report TR-91-10, Virginia Tech., Blacksburg, Virginia, 1991. URL http://hdl.handle.net/10919/19663.

31 D. Harman and G. T. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

32 D. Hawking and B. Billerbeck. Efficient in-memory, list-based text inversion. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 5:1–5:8, 2017.

33 H. S. Heaps. Storage analysis of a compression coding for document data bases. *INFOR: Information Systems and Operational Research*, 10(1):47–61, 1972.

34 S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.

35 S. Kim, T. Lee, S. w. Hwang, and S. Elnikety. List intersection for web search: Algorithms, cost models, and optimizations. *Proc. Conf. on Very Large Databases (VLDB)*, 12(1): 1–13, 2018.

36 D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software Practice & Experience*, 41(1):1–29, 2015.

37 D. Lemire, N. Kurz, and C. Rupp. Stream VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018.

38 N. Lester, J. Zobel, and H. E. Williams. Efficient online index maintenance for contiguous inverted lists. *Information Processing & Management*, 42(4):916–933, 2006.

39 N. Lester, A. Moffat, and J. Zobel. Efficient on-line index construction for text databases. *ACM Trans. on Database Systems*, 33(3):19.1–19.33, 2008.

40 J. Lin, L. Paniak, and G. Boerke. The performance envelope of inverted indexing on modern hardware. *arXiv:1910.11028v2*, 2019.

41 J. Mackenzie and A. Moffat. Examining the additivity of top-k query processing innovations. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1085–1094, 2020.

42 J. Mackenzie, M. Petri, and A. Moffat. Tradeoff options for bipartite graph partitioning. *IEEE Trans. on Knowledge and Data Engineering*, 2022. To appear.

43 A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 625–634, 2017.

44 A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. OSIRRC at SIGIR 2019*, pages 50–56, 2019.

45 A. Mallia, M. Siedlaczek, and T. Suel. An experimental study of index compression and DAAT query processing methods. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 353–368, 2019.

46 M. Mendoza, M. Marín, V. Gil-Costa, and F. Ferrarotti. Reducing hardware hit by queries in web search engines. *Information Processing & Management*, 52(6):1031–1052, 2016.

47 A. Moffat. Economical inversion of large text files. *Computing Systems*, 5(2):125–139, 1992.

48 A. Moffat and T. A. H. Bell. In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

49 A. Moffat and J. Mackenzie. Immediate-access indexing using space-efficient extensible arrays. In *Proc. Australasian Document Computing Symp. (ADCS)*, 2022. To appear.

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

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

52 G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):125.1–125.36, 2021.

53 S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations & Trends in Information Retrieval*, 3:333–389, 2009.

54 F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 222–229, 2002.

55 W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *Information Processing & Management*, 41(2):275–288, 2005.

56 K. Shoens, A. Tomasic, and H. Garćıa-Molina. Synthetic workload performance analysis of incremental updates. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 329–338, 1994.

57 A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 317–326, 2011.

58 A. Tomasic, H. Garćıa-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, page 289–300, 1994.

59 N. Tonellotto, C. Macdonald, and I. Ounis. Efficient query processing for scalable web search. *Foundations & Trends in Information Retrieval*, 12(4-5):319–500, 2018.

60 A. Trotman. Compressing inverted files. *Information Retrieval*, 6(1):5–19, 2003.

61 A. Trotman. Compression, SIMD, and postings lists. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 50.50–50.57, 2014.

62 A. Trotman, X. Jia, and M. Crane. Managing short postings lists. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 113–116, 2013.

63 H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995.

64 J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: Inverted list compression in memory. *Proc. Conf. on Very Large Databases (VLDB)*, 10(8):853–864, 2017.

65 H. E. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

66 J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6:1–6:56, 2006.