Structured Index Organizations for High-Throughput Text Querying

Vo Ngoc Anh and Alistair Moffat

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

Abstract. Inverted indexes are the preferred mechanism for supporting content-based queries in text retrieval systems, with the various data items usually stored compressed in some way. But different query modalities require that different information be held in the index. For example, phrase querying requires that word offsets be held as well as document numbers. In this study we describe an inverted index organization that provides efficient support for all of conjunctive Boolean queries, ranked queries, and phrase queries. Experimental results on a 426 GB document collection show that the methods we describe provide fast evaluation of all three querying modes.

1 Introduction

Inverted indexes are the preferred mechanism for supporting content-based queries in text retrieval systems, with the various data items usually stored compressed in some way [Witten et al., 1999, Zobel and Moffat, 2006]. For text documents, a document-level inverted index can typically be stored in under 10% of the space of the original documents.

However, different query modalities require that different information be held in the index, a need that creates tensions in the way that the index lists are organized. For example, phrase querying requires that word offsets be maintained in the index as well as document numbers, but those same word offsets represent unwanted decoding when conjunctive Boolean queries are being processed, or when ranked queries are being handled. One solution to this dilemma – and not as wasteful as it might at first appear – is to store two indexes, one of which contains word positions, and one of which contains only document numbers. Queries of different types can then be routed to the appropriate index, and overall average query throughput rates increased at the cost of additional disk space.

In this paper we first briefly summarize different types of index organization, and consider their ability to support fast evaluation of different types of query. A uniform framework is introduced that allows different organizations to be described in a succinct and unambiguous manner. The second part of the paper then describes a blocked interleaved inverted index organization that provides efficient support for all of conjunctive Boolean queries, ranked queries, and phrase queries. The proposed representation is a hybrid between a single index and a duplicated index, and exploits the best features of both. The paper concludes with experimental results on a 426 GB document collection show that the methods we describe provide fast evaluation of all of Boolean, ranked, and phrase queries.

2 Inverted Indexes

An inverted index for a collection of documents associates with each of its distinct term an *inverted list*, that stores a *pointer* for each document the term appears in. In simplest form a pointer consists of nothing more than an ordinal document number, meaning that the inverted list for a term t can be described as

$$\langle d \rangle^{f_t},$$

where f_t is the number of documents containing t; d is a document number; and the $\langle x \rangle^k$ notation indicates k repetitions of objects of type x. To resolve a query, the inverted lists for the query terms are fetched, and various operations performed on them, depending on the semantics assigned to the query operators.

There are a number of factors that determine what exactly is stored in the index, and how it is used to resolve queries. The rest of this section discusses these choices.

Pointer ordering: In a *document-sorted* index, the pointers in each inverted list are stored in increasing document order, allowing differences between consecutive term appearances to be stored as d-gaps, rather than absolute document numbers. Storing differences yields a more compact inverted file once compression techniques are applied, but requires that processing of each inverted list be in document number order.

An alternative is to use a *frequency-sorted* index [Persin et al., 1996] or an *impact-sorted* [Anh et al., 2001, Anh and Moffat, 2006b]. In the former structure, each inverted list is ordered in decreasing term frequency score $f_{d,t}$. In the latter, the pointers are ordered according to the *impact* value $\omega_{d,t}$, which is a small integer representing the overall contribution of term t to the score of document d, including the factor used to normalize for document length. In both arrangements, each index list is a sequence of groups of equally weighted pointers, and within each group the document numbers are sorted, and again stored as d-gaps. Experiments by Persin et al. [1996] and Anh et al. [2001] showed that these alternative representations can be stored in approximately the same space as a standard document-sorted index, but yield significantly faster processing of ranked queries.

Processing mode: Independent of the structure of the index or the type of query being handled, there are two main approaches to processing queries: the *document-at-a-time* model in which all inverted lists are simultaneously accessed, and $|q|$-way processing is carried out to handle a query q of $|q|$ terms; and the *term-at-a-time* model, in which only one inverted list is accessed at any given time, and a sequence of $|q|-1$ binary operations are performed. Recent work has tended to focus on term-at-a-time processing as applied to ranked querying, but it is not clear that term-at-a-time processing is fundamentally better than document-at-a-time processing. For example, Strohman et al. [2005] have

Table 1. Types of index and supported query modes

Type	d	$\omega_{d,t}$	pos.	Organization	Processing modes supported
D	Y	–	–	document-sorted	term-at-a-time, document-at-a-time
DS	Y	Y	–	document-sorted	term-at-a-time, document-at-a-time
				impact or frequency-sorted	term-at-a-time, score-at-a-time
DSP	Y	Y	Y	document-sorted	term-at-a-time, document-at-a-time

experimented with document-at-a-time orderings. As well as describing an improved index organization, this paper re-examines the issue of processing model, discussing the relative advantages of the two approaches, and comparing them experimentally on the same large 426 GB text collection.

For ranked querying with impact-sorted indexes, another approach, *score-at-a-time* has been mooted [Hawking, 1998, Anh et al., 2001]. In this mode, all of the term lists are open for reading, but are processed in decreasing score contribution order rather than in increasing document number order. Similar to the term-at-a-time approach, this method requires a set of accumulator variables, but facilitates dynamic query pruning; and the parts of each inverted list that are not required are potentially never read from disk.

Word positions and index levels: Another important way of categorizing an index is whether or not it includes within-document frequencies and within-document word positions. We distinguish between three levels of detail in regard to the index information stored and summarize the three respective index types in Table 1 along with the possible index organizations and the supported processing modes. As is listed in the table, a *Type D* index (*d*ocument numbers only) contains only document numbers, and is capable of supporting Boolean queries only. A *Type DS* (*d*ocument and *s*core) index stores both document numbers and either $\omega_{d,t}$ impact scores or $f_{d,t}$ within-document frequencies (from which $\omega_{d,t}$ values can be computed), or both, and can handle Boolean and ranked queries, but not phrase-queries. A *Type DSP* index (*d*ocument, *s*core, *p*ositions) contains document numbers, $\omega_{d,t}$ values (or $f_{d,t}$ values), and, for each document that the term appears in, a list of the ordinal word positions within the document at which the term appears. Type DSP indexes are capable of supporting all of the three mentioned query types (Boolean, ranked, phrase) and some other types such as proximity queries.

Table 1 also shows that impact- and frequency-sorting applies only to Type DS indexes, and is beneficial only when ranked queries are being performed. These indexes must be processed using a term-at-a-time strategy, or a score-at-a-time approach.

One of our aims in this investigation was to evaluate the extent to which – with suitable internal organizations applied to each inverted list – Type DSP indexes can be used to efficiently support Boolean and ranked queries. We sought to determine whether including positional information necessarily degraded querying performance for Boolean and ranked queries compared to the less voluminous Type D and Type DS indexes.

3 Interleaving

One key issue that has received little previous attention is that of *interleaving*, which also relates to the internal organization of each inverted list in a document-sorted index.

Pointer interleaving: In typical descriptions of Type DS inverted indexes, each document number (usually stored as a difference, or d-gap) is immediately followed by the corresponding $\omega_{d,t}$ or $f_{d,t}$ value. This arrangement can be described as:

$$\langle d, f_{d,t}\rangle^{f_t} .$$

In a Type DSP index, the word positions are usually described as being inserted immediately adjacent to the corresponding document number and $f_{d,t}$ value.. If, in addition, impact-based ranking is required (which is possible even if the index is document-sorted), the impact score is also included. The Type DSP index then has the form:

$$\left\langle d, \omega_{d,t}, f_{d,t}, \langle p\rangle^{f_{d,t}} \right\rangle^{f_t} ,$$

where p is a positional offset, and $\omega_{d,t}$ is a small integer [Anh et al., 2001].

We categorize such indexes as being *pointer-interleaved*, since the interleaving of different quantities is at the level of the individual pointers. Pointer-interleaving is the way that Type DS and Type DSP indexes are presented in textbooks (for example, [Witten et al., 1999]); in the research literature (for example, [Williams et al., 2004]); and in public domain software systems such as `Zettair` (available from `www.seg.rmit.edu.au`). What is not clear is whether or not commercial systems use pointer-interleaved indexes – such information is, of course, kept confidential.

Term interleaving: An alternative is to keep the various related parts of each inverted list in contiguous blocks, so that (in the case of a Type DSP index), the arrangement in each inverted list becomes:

$$\left\langle \langle d\rangle^{f_t}, \langle \omega_{d,t}\rangle^{f_t}, \langle f_{d,t}\rangle^{f_t}, \langle p\rangle^{\Sigma f_{d,t}} \right\rangle .$$

Non-interleaved indexing: The final option is for all of the document numbers for all of the pointers for all of the terms to be stored, then all of the ranking weights for all of the terms, then all of the within-document frequencies for all of the terms, and finally all of the word offsets for all of the terms. The arrangement is somewhat akin to having four separate inverted files, each storing a particular type of information, with four distinct disk pointers maintained in each entry in the vocabulary.

For completeness, our experiments also cover impact-sorted Type DS indexes, despite the fact that they are primarily designed for term-at-a-time and score-at-a-time processing of ranked queries. This kind of index has a specific structure where one impact score is shared by (in general) a sequence of documents, and allows fast querying because the amount of query-time computation is kept small.

Tradeoffs: There are many tensions between these representational and execution options. In a pointer-interleaved index, the information required for phrase and mixed querying is tightly clustered, and immediately available. In a term-interleaved index, the positional information is available within the terms' lists, but Boolean and ranked queries can be processed without the need to decode, or even bypass, the positional information. However the presence of the positional information at the end of every list might erode the effectiveness of buffering and caching strategies.

In a non-interleaved index, the positional information is fetched and decoded only when phrase queries are encountered in the input stream. Non-interleaved indexes also make it somewhat easier to apply compression, as longer sequences of values likely to be drawn from the same underlying probability distribution. This latter point is especially important in our system, which employs the binary-based `slide` compression technique, in which sequences of values of similar magnitudes are identified and represented compactly [Anh and Moffat, 2006a].

However, for phrase queries, non-interleaved indexes have the drawback of requiring information to be consolidated from several different access points, or *cursors*.

4 Block-Interleaved Indexes

To allow exploration of the trade-offs possible with different interleaving strategies, we introduce a hybrid approach called *block-interleaved* indexing

Groups and fixed-size blocks: The structure of a k-block interleaved index is given by:

$$\left\langle \langle d \rangle^k, \langle \omega_{d,t} \rangle^k, \langle f_{d,t} \rangle^k, \left\langle \langle p \rangle^k \right\rangle^{\lceil \Sigma f_{d,t}/k \rceil} \right\rangle^{\lceil f_t/k \rceil}$$

Each sub-unit $\langle x \rangle^k$ for some kind of value x is a k-*block*, and each unit of k complete pointers is a *group*. That is, each inverted list is built up as a sequence of one or more groups, and within each group there are four or more k-blocks. Figure 1 gives an example list that is stored as two groups of (at most) $k = 4$ pointers; and within each group, all of the values are stored as k-blocks. Note that the set of positional offsets $\langle p \rangle^k$ are also in blocks of k, and might cross pointer boundaries. In this structure a group is a streamlined bundle of k pointers.

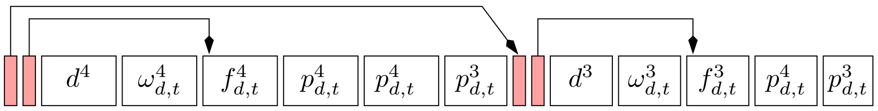

Fig. 1. An example block-interleaved index structure for an inverted list of $f_t = 7$ pointers, using $k = 4$. The list is organized in two groups. The first group contain $k = 4$ pointers, and the second group holds 3 pointers. Within the groups, all of the data items are stored in blocks. In the example, the sum of the $f_{d,t}$ values in the first group is assumed to be 11; and in the second group, 7. The shaded items are internal access structures to facilitate skipping past unneeded data.

The routines that manipulate k-blocks are optimized to support an interface that fetches and decodes the next (as many as) k values from an inverted list. A key benefit of this approach is that the decoding buffer is exactly bounded at k values for each of the blocks that are required. None of the query modes require that more than four blocks be active per term, and thus that each term requires decode buffer space of $4k$ words.

Skip pointers: The shaded items in Figure 1 are internal *skips*, that allow sections to be stepped over and not decoded. Skips are typically represented as byte or bit increments that lead to the start of some future codeword. Each group in a block-interleaved list has two skip pointers: a *group skip* that references the start position of the next group, and a *block skip* that provides access within the group, allowing the k-block of $\omega_{d,t}$ values to be stepped over, so that the set of $f_{d,t}$ values can be directly accessed. The blocks of positional offsets are never accessed without the block of $f_{d,t}$ values being retrieved first; and so there is no need for a skip to the start of the positions.

The group skips described here are similar to the skips used by Moffat and Zobel [1996] to speed up the term-at-a-time processing of Boolean and ranked queries. Similar improvement can be expected for our group skips. On the other hand, the block skips have a different function, and they allow bypassing of blocks of values that are not required in processing this query, for example, the positional information when Boolean queries are being processed. The skips of Moffat and Zobel are to facilitate searching for candidates, perhaps as a result of dynamic query pruning and accumulator limiting, and their index is stored in a pointer interleaved manner. Though it is possible, no pruning of ranked queries occurs in any of the experiments reported in this paper, so that unfair comparison between different processing modes is avoided.

Processing queries: A further issue worth elaboration is that of how queries are processed. Taking the simplest possible example, and presuming a document-sorted index and a conjunctive Boolean query, the task at hand is to identify the set of document numbers that appear in all of the terms' inverted lists.

If a term-at-a-time processing strategy is adopted, then the standard mechanism for doing this is to process the terms in increasing f_t order, using the first term to establish a set of *candidate* answers C, and then for each subsequent term t, checking each of the candidates against t's inverted list:

```
1: open the inverted list for the term $t_1$ with the smallest $f_t$.
2: set $C \leftarrow copy\_list(t_1)$.
3: for $i \leftarrow 2$ to $|q|$ do
4:     open the inverted list for term $t_i$.
5:     for each candidate $c \in C$, in increasing order, do
6:         set $d \leftarrow seek\_list\_value(t_i, c)$.
7:         if $d > c$ then
8:             set $C \leftarrow C - \{c\}$.
9:             if $|C| = 0$ then
10:                return the empty set.
11: return $C$.
```

The key operation performed is that of "*seek_list_value*(t, c)", which seeks forwards in the inverted list of term t until a document number greater than or equal to c is encountered. If c is actually found, it is retained as a candidate; if the next document number d is greater than c, then c is removed as a candidate.

If document-at-a-time processing is being carried out, the set of candidates is not required, but all of the terms' lists need to be simultaneously open:

```
1:  set $C \leftarrow \{\,\}$.
2:  for $i \leftarrow 1$ to $|q|$ do
3:      open the inverted list for term $t_i$.
4:      set $d_i \leftarrow$ next_list_value$(t_i)$.
5:  while all lists have pointers remaining do
6:      set $d \leftarrow \max\{d_i \mid 1 \le i \le |q|\}$.
7:      for $i \leftarrow 1$ to $|q|$ do
8:          set $d_i \leftarrow$ seek_list_value$(t_i, d)$.
9:          set $d \leftarrow \max\{d, d_i\}$.
10:     if $\min\{d_i \mid 1 \le i \le |q|\} = d$ then
11:         set $C \leftarrow C + \{d\}$.
12:         for $i \leftarrow 1$ to $|q|$ do
13:             set $d_i \leftarrow$ next_list_value$(t_i)$.
14: return $C$.
```

Note that the locus of activity does not pause at every pointer in every inverted list. Instead, the focus leap-frogs down the set of lists, using *seek_list_value* operations to try and catch each list's activity zone up to the current document number in the list that to date has progressed the furthest. In particular, the costly "max" and "min" operations over the full set of $|q|$ current document numbers are relatively infrequent, and occur at most once for each of the items in the shortest inverted list. They do not occur for every pointer in every list, as would be the case in a heap-based merging process.

In a block-interleaved index, the *seek_list_value* operation still proceeds as a sequential decoding of a k-block of d-gaps, but is done within the routines handling the inverted list. They transparently use the block skip pointer in a group to access the next group if the sum of the remaining d-gaps in the current block does not reach the specified target value. While linear-time in its underlying operation, the nature of the compression process means that this operation is extremely fast, and only the first k-block of values is read and decoded in any groups that are otherwise completely skipped.

Also worth noting is that the same basic operations are used to implement phrase queries. The query is initially handled as a Boolean one, and only when a document is determined to contain all of the query terms is any access made to any of the k-blocks storing the positional information. On the other hand, the current implementation of ranked querying does require that every d-gap and every $\omega_{d,t}$ impact value be processed – dynamic pruning issues have not yet been explored.

Other blocking issues: The parameter k determines the amount of compressed information decoded in each access to the pointers in a group, and reflects the amount of decompressed information that is maintained at any point in time. Another important parameter is the unit of access to compressed information – the size of the buffers into which inverted lists are read when required by the decoding routines. Our system does not read the whole of each inverted list in a single operation, since doing so requires compressed buffers of indeterminate (and variable) size. Instead, each inverted list is read "on demand" in compressed blocks of a fixed size, and input buffers are bounded. The drawback is that multiple seeks may be required to access a given list in its entirety, even in a term-at-a-time processing model; but in a disk-block-based file system this is likely anyway, and our arrangement simply acknowledges that reality.

Setting a size to the list buffers again involves competing tensions. Large buffers reduce the number of seek operations, but may add unnecessary decoding costs. On the other hand, use of overly short logical blocks implies multiple accesses even when relatively short inverted lists are being processed. In the experiments reported below the compressed block size (the unit read from disk) was fixed at 8 kB; and the logical block size (the amount of data decoded in each call to the decoding routines, the value k in Figure 1) was initially set at 8,192 integers.

5 Experimental Evaluation

The various implementation options have been tested and compared on a large collection of typical web data – the 426 GB `GOV2` collection. This collection was created as part of the TREC initiative, see `trec.nist.gov`, and was drawn from an early-2004 crawl of the `.gov` domain. It contains approximately 25 million documents.

Queries: Two query sets, `Q1000` and `Q321`, are drawn from the set of 50,000 real-life queries used in the 2005 TREC Terabyte Track experiments. The former set consists of the first 1,000 queries from the superset; the latter set contains only those queries of `Q1000` that contain at least two terms and have at least one answer when considered as phrase queries against the `GOV2` collection. There are 321 queries that satisfy these two requirements. Over the two sets, the average numbers of terms per query are 2.79 and 2.41; the average number of conjunctive Boolean answers are approximately 79×10^3 and 72×10^3, respectively.

Baseline: To establish a reference point for query speed, we started with conjunctive Boolean querying. The second data column of Table 2 shows the average query processing rate (queries per second) for Boolean queries, document-at-a-time processing, the `GOV2` collection and `Q1000` queries, and several different index types.

As was anticipated, the best Boolean querying performance is achieved by a Type D index. Use of a pointer-interleaved Type DS index slows query handling by around 40%, and a pointer-interleaved Type DSP index slices more than 90% off the performance. A term-interleaved Type DS index allows the original level of performance to be regained, but even with a Type DSP term-interleaved index there is some loss of throughput.

The next column of Table 2 shows the throughput rate at which the `Q1000` queries can be processed as ranked queries. All of the Type DS indexes provide comparable performance, at throughput rates around half of what can achieved when the same queries are treated as being Boolean conjunctions. Querying rates with a Type DSP index are again slow if a pointer-interleaved index is used. Recall that these throughput rates represent exhaustive document-at-a-time processing without any form of query pruning.

Table 2. Performance on a single 2.8 GHz Intel Xeon with 1 GB RAM and the collection `GOV2`: index size and querying throughput for different query types with document-at-a-time processing using two query sets, measured as queries per second. The logical access unit in to the compressed streams is $k = 8{,}192$ integers.

Index arrangement		Index Size	`Q1000`			`Q321`		
		(GB)	Boolean	Ranked	Phrase	Boolean	Ranked	Phrase
Type D		**5.05**	**5.73**	–	–	**5.24**	–	–
Type DS	pointer-interl.	9.21	3.23	2.46	–	3.47	**3.02**	–
	term-interl.	**6.85**	5.71	**2.49**	–	4.99	2.80	–
	non-interl.	**6.85**	5.52	2.46	–	5.12	2.69	–
Type DSP	pointer-interl.	47.18	0.48	0.46	0.46	0.58	0.58	0.60
	term-interl	**36.13**	5.35	2.33	**0.47**	4.95	2.62	**0.64**
	non-interl.	**36.13**	5.50	2.37	**0.47**	5.02	2.56	**0.64**

The fourth column of Table 2 shows the throughput rate at which the `Q1000` queries can be processed as phrase queries. A Type DSP index is required, and the low throughput levels achieved reflect processing of a large volume of compressed data. All types of interleaving give comparable speeds. Note that there are many queries in `Q1000` that are just a single term, and many multi-term queries for which there are no answers.

The last three columns of Table 2 show query throughput rates for the query subset `Q321`, for which every query has at least one Boolean answer and one phrase answer. Similar trends in performance are observed.

Index size: The first data column in Table 2 shows the size of the various inverted indexes used in these experiments. When compressed, the document components account for space equivalent to only a little over 1% of the initial collection; the $\omega_{d,t}$ components for a further 0.5% in a term-interleaved setting (which allows "runs" of like values to be exploited by the compression regime); and then the word positions account for a further approximately 6%. The total Type DSP term-interleaved index can be stored in less than 40 GB, or under 10% of the size of the data. Note also that with the particular compression mechanism used the pointer interleaved index is more expensive to store than the term-interleaved and non-interleaved variants, because pointer interleaving makes the values in each list less locally homogeneous [Anh and Moffat, 2006a].

Block-interleaved indexes: Table 3 shows the performance of Type DS and Type DSP block-interleaved indexes, in an experimental setting comparable to that used to obtain the results in Table 2, but with a logical block size of $k = 1{,}048{,}576$. Even with word positions included (the Type DSP index), all of Boolean, ranked, and phrase queries are processed at similar or better rates to those shown in Table 2.

Nor is there any penalty in terms of index size – the change to block interleaving involves an index cost for the `GOV2` collection 1.59% for the Type DS index, and 8.44% for the Type DSP index.

Choosing a block size: Figure 2 shows how query throughput is affected by the choice of the logical block size parameter k. Because one of the key query processing costs

Table 3. Querying throughput for block-interleaved indexes and two query sets. The logical access unit in to the compressed streams is the same as the blocksize, $k = 1{,}048{,}576$.

Index arrangement	Q1000			Q321		
	Boolean	Ranked	Phrase	Boolean	Ranked	Phrase
Type DS block-interl.	**5.76**	**2.56**	–	5.16	**3.07**	–
Type DSP block-interl.	5.15	2.41	**1.32**	4.76	2.88	**0.69**

is disk seek times, small values of k are relatively inefficient. At the left of the graph, when $k < 1{,}000$, performance is very similar to that obtained from a pointer-interleaved Type DSP index, because multiple blocked groups fit within each 8 kB physical access block. Savings appear when k is 10,000 or more, because groups now span more than one disk block, and the skip pointers mean that some of the seeks are eliminated. The best performance for all querying modalities arises at k or around one million. With this value of k only a small fraction of the inverted lists are split into more than one group, meaning that the majority of the index is stored as if it were term-interleaved. Nevertheless, buffer sizes and thus caching costs are controlled at the same time as all of the throughput gains of the term-interleaved index are attained.

Processing modes and speed: All of the experiments reported in the previous section are for document-at-a-time evaluation, which has the advantage of requiring per-query execution space proportional to the number of answers being generated rather than proportional to the length of any of the inverted lists being processed. For example, Heaps [1978, Chapter 6] describes the document-at-a-time mechanism, as do Turtle and Flood [1995] and Strohman et al. [2005]. But Section 2 mentioned two other ways in which queries can be evaluated: the term-at-a-time, and the score-at-a-time approaches. In particular, Witten et al. [1999] present query processing using the term-at-a-time paradigm, for both Boolean and ranked queries. Kaszkiel et al. [1999] evaluate both of those

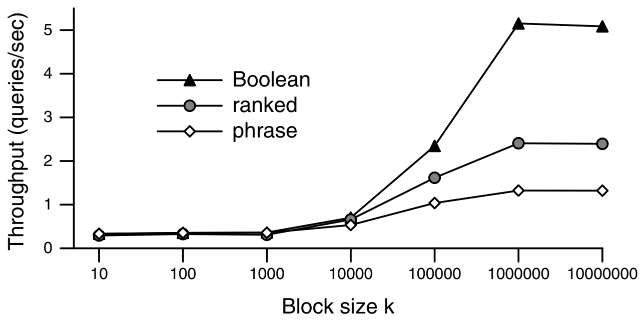

Fig. 2. Query throughput rate as a function of the logical block size parameter k, using collection GOV2, a Type DSP block-interleaved index, query set Q1000, document-at-a-time processing, and three different query types.

Table 4. Querying throughput for different query processing modes using query sets `Q1000` and `Q321`, with other details as for Table 3. The two pointer-interleaved indexes are document-sorted.

Index arrangement	Index Size (GB)	`Q1000` Ranked	`Q321` Ranked
Type DS pointer-interl. document-at-a-time	9.21	2.46	3.02
Type DS pointer-interl. term-at-a-time	9.21	2.35	2.29
Type DS impact-sorted score-at-a-time	5.99	**4.12**	**3.50**

strategies in the context of passage retrieval, and conclude that document-at-a-time is superior when the number of terms is small, but that term-at-a-time is to be preferred when the number of terms in the query is more than around 3–5. Kaszkiel et al. also describe a hybrid mechanism that processes rare terms in document-at-a-time mode, then the remainder in term-at-a-time mode.

The score-at-a-time approach of Anh et al. [2001] represents a hybrid between the term-at-a-time and document-at-a-time approaches. (Anh et al. also showed that the use of integer impacts and the avoidance of floating point computations allowed fast processing or ranked queries, and an integer-based similarity calculation is used in all of the experiments reported here.) To round out our experiments, we thus carried out a final set of runs using Type DS impact-sorted indexes and the same similarity computation, and score-at-a-time processing to handle ranked queries. Table 4 shows the results.

The score-at-a-time regime provides faster query processing than either document-at-a-time or term-at-a-time processing, primarily because of the way the index is structured. In an impact-sorted index, each impact score is followed by a sequence of d-gaps representing documents that all share that impact, and so each pointer that is processed requires $1+\epsilon$ values to be decoded (the ϵ being the shared impact value) rather than the 2 values per pointer that are decoded in a pointer-interleaved document-sorted index. The index is also slightly smaller, and for the `GOV2` collection the Type DS impact-sorted index occupies just 1.42% of the source files. Even faster impact-ordered processing is possible if dynamic query pruning is employed [Anh and Moffat, 2006b].

6 Conclusion

We have categorized inverted index structures in a number of ways, including with respect to the information that they contain, the way that information is organized within the index, and the way that the index is used to resolve queries. We have also carried out comprehensive experiments using a 426 GB collection of web documents, and a stream of 1,000 real-world queries.

At one level, the results we have achieved are relatively "intuitive" – it is hardly surprising that non-interleaved or term-interleaved index structures give faster Boolean query throughput than do pointer-interleaved structures. But the extent of the throughput difference is notable, and it is clear from our results that pointer interleaved structures should not be considered for practical implementation, despite their simplicity. In this respect the careful implementation and experimentation reported in this paper represents a significant and tangible contribution. In addition, the block-interleaved index organization we have introduced allows buffering costs to be controlled, without sacrificing any querying speed. Block-interleaved indexes provide support for both document-at-a-time and term-at-a-time processing, and establish a query evaluation framework against which other proposed developments can be measured.

A key area for ongoing investigation is in the area of dynamic pruning techniques, and document-at-a-time query processing. All of the results presented here are for exhaustive evaluation of queries; further throughput improvements are likely to be possible when pruning techniques appropriate to block-interleaved indexes are tested.

Acknowledgment. This work was supported by the Australian Research Council, the ARC Special Research Center for Perceptive and Intelligent Machines in Complex Environments, and by the NICTA Victoria Laboratory.

References

V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42, New Orleans, Louisiana, Sept. 2001. ACM Press, New York.

V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, June 2006a.

V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, Aug. 2006b. ACM Press, New York. To appear.

D. Hawking. Efficiency/effectiveness trade-offs in query processing. *ACM SIGIR Forum*, 32(2): 16–22, Sept. 1998.

H. S. Heaps. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, 1978.

M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17(4):406–439, Oct. 1999.

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

M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, Oct. 1996.

T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In G. Marchionini, A. Moffat, J. Tait, R. Baeza-Yates, and N. Ziviani, editors, *Proc. 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 219–225, Salvador, Brazil, Aug. 2005. ACM Press, New York.

H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(1):831–850, Nov. 1995.

H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, 22(4):573–594, 2004.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.

J. Zobel and A. Moffat. Inverted files for text search engines. *Computing Surveys*, 2006. To appear.