Hybrid Bitvector Index Compression

Alistair Moffat
Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia 3010
alistair@csse.unimelb.edu.au

J. Shane Culpepper
NICTA Victoria Research Laboratory,
Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia 3010
shanec@csse.unimelb.edu.au

Abstract *Bitvector index representations provide fast resolution of conjunctive Boolean queries, but require a great deal of storage space. On the other hand, compressed index representations are space-efficient, but query evaluation tends to be slower than bitvector evaluation, because of the need for sequential or pseudo-random access into the compressed index lists. Here we investigate a simple hybrid mechanism that stores only a small fraction of the inverted lists as bitvectors and has no or negligible effect on compressed index size compared to the use of byte codes, but improves query processing throughput compared to both byte coded representations and entirely-bitvector arrangements.*

Keywords Index compression, bitvector, byte code, intersection algorithm.

1 Text search

Document retrieval systems typically make use of an inverted index in order to provide fast keyword-based search, see, for example, Witten et al. [1999] and Zobel and Moffat [2006]. In such an index, an inverted list is stored for each term that appears in the collection, containing the ordinal identifiers of the documents in which that term appears. To process a conjunctive Boolean query, the inverted lists corresponding to the query terms are fetched from disk, and their set intersection computed. Some forms of "ranked" query, where the ranking component consists entirely of static precomputed score elements such as PageRank, can also be handled the same way – the document collection is permuted so that document identifiers are assigned in decreasing static score order, and then "top-k ranked queries" are resolved by identifying and presenting the first k matching conjunctive Boolean answers, without any further ranking step being applied.

A wide variety of representations have been developed to store the inverted lists, each of which is a set of *document pointers* representing a subset of the integers $1 \ldots n$, where n is the number of documents in the collection. For example, a set of f elements in the range $1 \ldots n$ can be stored in $f\lceil \log_2 n \rceil$ bits using a simple binary code; and can be stored in approximately $f(1.5 + \log_2(n/f))$ bits if a Golomb or Rice code is applied to the set of d-gap differences between consecutive items in the ordered set. Witten et al. [1999] provide details of these representations. For typical document collections, in which the majority of terms appear in just a few documents, but the majority of the pointers stored are for terms that appear in many documents, compressed representations (of which Golomb and Rice codes are examples) typically save as much as 70–80% of the space that would be required by a simple binary code.

Just as there are different ways in which the inverted lists can be stored, there are also different ways in which they can be manipulated in order to compute intersections. For example, if the inverted lists are stored compressed using a Golomb code, and no auxiliary information of any kind is maintained, then the intersection of two lists of $f_1 \le f_2$ elements requires $O(f_1 + f_2) = O(f_2)$ time, since there is no alternative to sequential decompression of both lists. On the other hand, if the lists are stored using the more space-costly fixed-width binary codes, and individual elements can be accessed and inspected in $O(1)$ time, then set intersection can be computed in $O(f_1 \log(f_2/f_1))$ time via a search-dual of the Golomb code, as described by Hwang and Lin [1972]. Other combinations of representation and intersection method are discussed by Culpepper and Moffat [2007].

When multiple lists are to be intersected, rather than just two, another dimension of choice is introduced. One possibility is to compute the intersection by per-

Proceedings of the 12th Australasian Document Computing Symposium, Melbourne, Australia, December 10, 2007.

forming a sequence of pairwise merges in which the shortest initial list is taken as a "pivot", and repeatedly intersected against another of the initial sets, in a *set versus set*, or svs, approach. Any method for the pairwise merging of sets can be employed, including the simple linear-time sequential method. The alternative is to holistically open all of the sets simultaneously, and intersect them in an interleaved manner, with a multiway operation being used to generate the list of document identifiers that appears in every one of the input lists. In this case, different intersection methods are possible, including *adaptive* variants that are sensitive to the interactions between the lists being joined [Demaine et al., 2000, Barbay et al., 2006].

This paper describes a hybrid bitvector representation for inverted indexes, and shows experimentally that it provides fast and compact execution of typical conjunctive Boolean queries compared to previous approaches.

2 Experimenting with intersection

In order to explore different inverted list structures and intersection algorithms, an experimental testbed was created in which a stream of conjunctive queries was processed against the index of the 426 GB `gov2` collection (see `trec.nist.gov`). Words that appeared only one or twice in the 25,205,181-document collection were assumed to not have their own index lists, and the result was an index for 19,783,975 distinct words, with each of those lists containing on average 307.6 ordinal document numbers.

The query stream contained 27,004 queries of average length 2.73 terms extracted from an operational "live" web query stream as being ones that were applicable to the experimental collection, by virtue of their having a whole-of-web top-3 answer (at the time they were issued) within the `.gov` domain. In total, 15,208 distinct terms appeared in the query set, a very small fraction of the terms in the collection. Culpepper and Moffat [2007] give more details of the experimental arrangements and of the query set.

To measure CPU times, the set of index lists required by the first query in the sequence was read in to memory while the execution clock was halted; then the clock was started and the intersection of those lists computed five times, to generate an "answer" list of ordinal document numbers; then the clock was halted again while the data for the second query was fetched; and so on. At the same time as these "average over five" times were being recorded, the amount of index data (in megabytes) transferred from disk to memory during query evaluation was noted, and used later to obtain a per-query average data volume. All of the experiments were carried out on a dual 2.8 Ghz Intel Xeon with 2 GB of RAM, twelve 146 GB SCSI disks in a RAID-5 configuration, and running Debian GNU/Linux.

Figure 1, adapted from Culpepper and Moffat [2007], summarizes some of the experiments

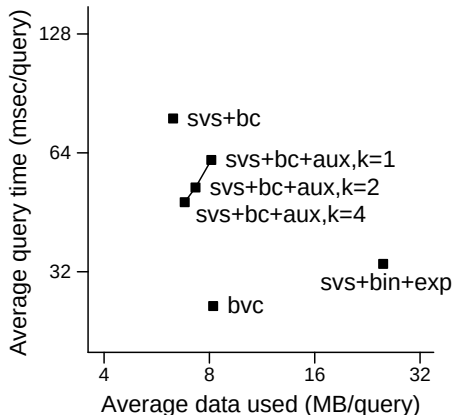

Figure 1: *Space versus CPU cost of different ways of computing conjunctive Boolean queries using a 2.8 Ghz Intel Xeon with 2 GB of RAM, adapted from Culpepper and Moffat [2007]. The horizontal axis shows the average per-query data volume required to process a set of 27,004 queries; the vertical axis shows the average time taken over the same query set. Each of the marked points represents a combination of compression technique and intersection algorithm. The point marked* bvc *makes use of a bitvector representation and computes intersections using word-at-a-time "*AND*" operations.*

undertaken using the test harness. In that work we were interested in the extent to which a small auxiliary index in each inverted list (the annotation aux on three of the data points) allowed improved trade-offs between time and space. The auxiliary index in this method was stored uncompressed, and provided a set of access points in to the byte coded compressed inverted lists, so that the forwards searching operation required by the svs approach could be supported via pseudo-random access. A parameter k was used to balance the additional cost of the auxiliary index against the desire to keep the access points close together. Compared to a svs method implemented using a binary-coded index representation; and compared to the svs method when the data was stored compressed using standard byte codes (annotation bc, and described in more detail below), the auxiliary index approach did indeed offer an interesting compromise between speed (average query time, on the vertical axis) and space (average data volume processed, on the horizontal axis).

3 An interesting observation

Another interesting point in Figure 1 – and the basis for the further exploration that is described in this paper – is the one marked bvc. A bitvector is a very simple way of storing a set of f elements drawn from a universe $1 \ldots n$, with an n-bit array constructed in which the i th bit is set if and only if i is a member of the set. Bitvectors are a very expensive way of storing sparse sets, in which $f \ll n$. On the other hand, they have the redeeming virtue of providing $O(1)$-time lookup, meaning that a set of f_1 candidate answers can be checked against

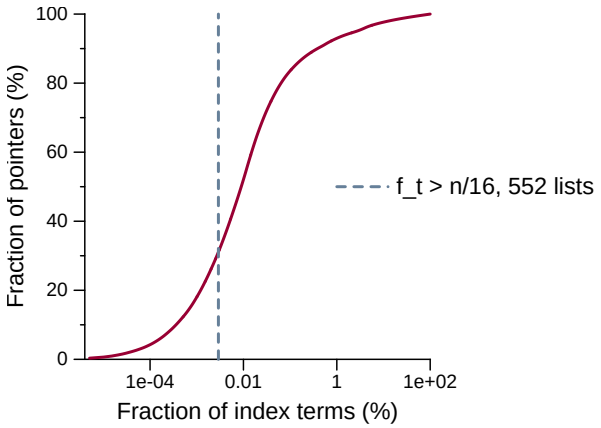

Figure 2: *Terms versus pointers: a small fraction of the terms in the collection are responsible for the great majority of the index pointers. For example, the vertical dotted line indicates that approximately* 30% *of the pointers in the index appear in the set of* 552 *inverted lists that each contain more than* $25{,}205{,}181/16 = 1{,}575{,}323$ *document identifiers.*

a second set of f_2 items in $O(f_1)$ time, once the n bits of the vector have been read. In addition, if two bitvectors are to be intersected, 32 or 64 bits at a time can be processed using whole-of-word AND operations.

The latter of these two processing modes was used to generate the bvc data point for Figure 1, and it was unsurprising that bitvectors provided fast intersection operations. What had not been expected was that the volume of data required to process the queries using a bitvector representation was also comparable with techniques that involved compression. Storing a whole index using a bitvector per index term is exorbitantly expensive, approximately 3 MB per index list in the gov2 test collection described above, making a total of nearly 60 TB for the 19 million terms indexed. But the actual queries processed tend not to be against sparse lists. Even the two-word queries in our test set typically had one word that appeared in 5% or more of the documents in the collection, and once the query reached four or more terms, on average one (or more) of the supplied query terms appeared in more than half of the documents in the collection [Culpepper and Moffat, 2007].

The bvc point in Figure 1 thus leads to an obvious question: is there some compromise arrangement that avoids the very high disk storage cost of the full bitvector index, but retains its efficiency in terms of query evaluation speed, and in terms of data volume transferred in order to process queries.

Figure 2 shows why such a hybrid approach is attractive. To construct this graph, the terms of the gov2 collection were ordered by decreasing document frequency, and then a cumulative count of pointers calculated, based on that ordering. For example, the most frequent term contributes 20,461,040 pointers, or 0.33% of the 6,086,023,363 pointers making up the index; the second most frequent term adds another 18,964,349 pointers, and so on. Both axes of the graph are expressed as percentages, but the horizontal axis is plotted logarithmically.

What is clear from Figure 2 is that a very small fraction of the collection's terms account for a very large fraction of the index pointers. Just 0.01% of the terms account for more than 50% of the pointers, and if 1% of the terms are handled via bitvectors, more than 90% of the pointers in the index are covered.

4 A hybrid approach

The use of Golomb and Rice codes provide good compression for typical index data, in no small part because they are sensitive to the ratio f/n (see Witten et al. [1999] for a description of the Golomb code). For example, when more than around 38% of the documents in a collection contain some term, the Golomb parameter b that determines the codewords will be 1, and the effect is that of a Unary code. In such a case, the result of the coding exercise is a bitvector.

Another standard coding method for index lists (or rather, the differences between successive values in them) is via *byte codes*. In a byte code, each codeword is a multiple of eight bits long, so that all of the codewords consist of an integral number of bytes. The simplest byte code, denoted bc here, uses a single bit in each byte of the codeword to indicate whether or not this is the final byte in the value, and uses 7, or 14, or 21, and so on, bits to store the integer value in question. Using the bc approach, document pointer differences between 1 and $2^7 = 128$ are stored in a single byte; differences between 129 and $2^7 + 2^{14} = 16{,}512$ are stored in two-byte codes; and differences from 16,513 to $2^7 + 2^{14} + 2^{21} = 2{,}113{,}664$ are stored as three-byte codewords. Scholer et al. [2002] examine the use of byte codes in inverted file indexing, and Brisaboa et al. [2003] and Culpepper and Moffat [2005] describe alternative byte codes with additional properties.

A byte code compression scheme was used in the bc-annotated systems shown in Figure 1. Given that the minimum codeword length in any byte code is 8 bits, and thus that a set of f elements in the range $1 \ldots n$ must consume at least $8f$ bits when coded, even after differences are taken, it is clear that a bitvector representation (in which n bits are consumed) is more compact than a byte code for any terms for which $f > n/8$. Figure 2 shows that point for the gov2 collection – the dashed vertical line separates the index lists for which a bitvector representation must be more economical (in terms of storage) from those for which a byte code is most likely to be the more economical option. As was already noted, that implies that fully half of the pointers in the index are more economically coded via a bitvector than via byte coded differences.

Storing the index list for a term t as a bitvector whenever the document frequency f_t of term t satisfies $f_t > n/8$ will thus reduce the size of a byte coded inverted index. It is also possible to allow more than the

Method	Bitvector terms	Size (GB)
Byte coded	0	7.41
Hybrid, $f_t > n/8$	188	6.97
Hybrid, $f_t > n/10$	277	7.00
Hybrid, $f_t > n/12$	367	7.07
Hybrid, $f_t > n/16$	552	7.31
Hybrid, $f_t > n/20$	775	7.67
Hybrid, $f_t > n/24$	982	8.05
Hybrid, $f_t > n/32$	1,382	8.88

Table 1: *Cost of storing a* gov2 *index when terms appearing in more than a specified fraction of the documents are stored as bitvectors rather than as Byte coded inverted lists.*

minimum number of the inverted lists the flexibility of using a bitvector. Table 1 shows the cost, in gigabytes, of storing the gov2 index ($19{,}783{,}975$ terms, and $6{,}086{,}023{,}363$ pointers) using byte codes alone, and then using a bitvector hybrid approach with different cutoff fractions. As can be seen, cutoffs as small as $f_t > n/32$ still result in a gov2 index that is only a little larger than the byte coded one, and when the cutoff is $n/16$ or less, the index is smaller.

Many of these frequently-occurring terms are ones that are, at face value, unhelpful during querying. Indeed, in the past, they may well have been *stopped*, to reduce the space consumption of the index, rendering the hybrid scheme proposed here somewhat moot. However, modern retrieval and web search systems index all words and numbers, with queries such as "to be or not to be", "Dr Who", and "11 September 2001" being examples that show why complete coverage is necessary. The statistics quoted earlier in connection with the test query stream show that common words do indeed occur in typical web search queries.

Processing queries: Method One

Given the hybrid index, the obvious question now is how best to use it to resolve conjunctive Boolean queries. We experimented with two query processing approaches. In the first, queries are executed as follows:

1. All query terms are located in the vocabulary.
2. The set of terms with byte coded inverted lists, if any, are intersected using the svs approach, to yield a set of candidate answers C. If C becomes empty at any stage, then there are no answers to the query, and processing terminates.
3. The set of terms with bitvectors, if any, are intersected to get a bitvector B that represents their conjunction.
4. If there were no bitvector terms, then C can be output as the answer to the query.
5. If there were no byte coded terms, then B represents the answer, and is converted into a set of document numbers by locating all of the subscripts d for which $B[d] = 1$.
6. Otherwise, for each $d \in C$, if $B[d] = 1$, then d is output as an answer to the query.

That is, all of the byte coded terms are intersected first; then, if necessary, all of the bitvector-represented terms are intersected to get a combined bitvector; and then, if necessary, the set of candidates indicated by the byte coded terms are checked against the outcome of the bitvector merge.

Processing queries: Method Two

In the second approach to query processing, the bitvector terms are incorporated incrementally via a sequence of lookups once a set of candidate answers has been established:

1. All query terms are located in the vocabulary.
2. If there are no byte coded terms, then the bitvector terms are intersected as in Method One, and returned as the answer.
3. Otherwise, the set of terms with byte coded inverted lists are intersected using the svs approach to yield a set of candidate answers C. If C becomes empty at any stage, then there are no answers to the query, and processing terminates.
4. For each bitvector term, every element still left in C is checked against that bitvector, and retained in C only if it appears in that bitvector. If C becomes empty at any stage, then there are no answers to the query, and processing terminates.
5. When all bitvector terms have been processed, C is the list of answers.

This method avoids intersecting bitvectors once the number of viable candidate answers is small. Instead, it builds on the fact that bitvectors support random-access membership queries, and the fact that it is relatively cheap to check a small set of candidates against a bitvector by direct probing of the relevant bit positions. That is, when the size of the set of candidates C is small, which it almost certainly must be at the conclusion of the byte coded phase, it should be faster to check individual candidates against each bitvector than to AND all of the bitvectors terms together.

5 Experiments

We used the same test harness as for our previous experiments [Culpepper and Moffat, 2007], so as to be able to compare results. As was the case in that work, a set of $27{,}004$ "real" queries derived from a search log were executed, and CPU time and data transfer volume measured on a 2.8 Ghz Intel Xeon with 2 GB of RAM.

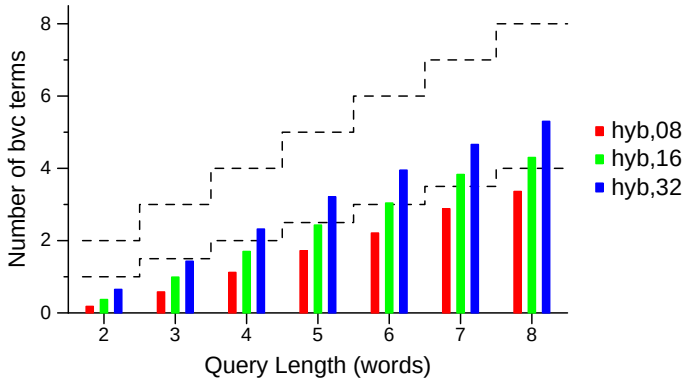

Figure 3: *Fraction of the terms in each query processed using bitvectors, for different index construction thresholds. The upper dashed line shows the total number of terms in the query; and the lower dashed line shows half of the terms. When $f_t > n/32$, on average half or more of the terms in queries of length three and greater are processed as bitvectors, either through whole-of-word* AND *operations (Method One), or via direct bit lookups (Method Two).*

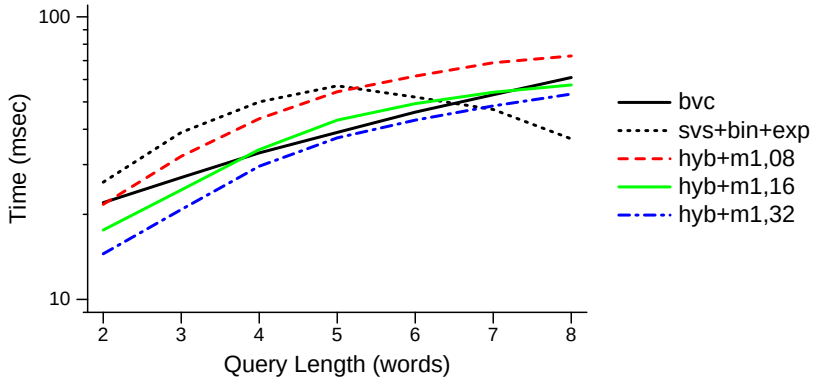

Figure 4: *Time to calculate answers to conjunctive queries using a hybrid bitvector representation, Method One processing, and three different bitvector cutoffs. The solid black line represents the pure bitvector approach, and the dotted black line shows a pure* SVS *approach using a binary-coded index and exponential search. When $f_t > n/32$ is the threshold point, the hybrid approach is faster than the pure bitvector approach for all tested query lengths.*

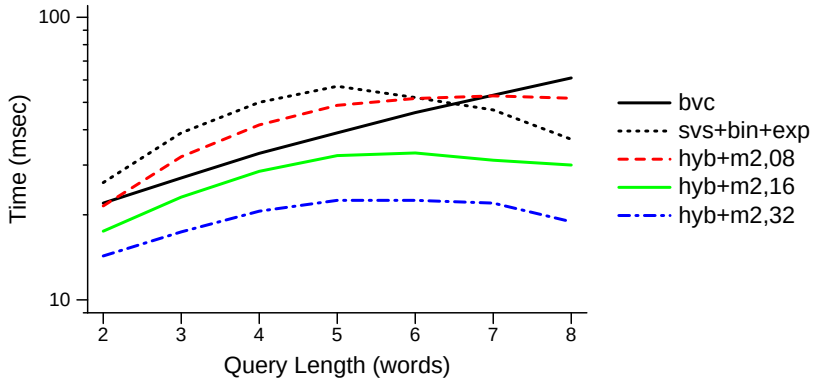

Figure 5: *Time to calculate answers to conjunctive queries using a hybrid bitvector representation, Method Two processing, and three different bitvector cutoffs. The solid black line represents the pure bitvector approach, and the dotted black line shows a pure* SVS *approach using a binary-coded index and exponential search. The $f_t > n/32$ hybrid index provides significantly faster query processing than all other methods tested, across all query lengths.*

Figure 3 provides evidence in support of our hypothesis that the hybrid approach is preferable to a purely byte coded index. When the queries are analyzed based on their length in terms, the prevalence of common terms in queries becomes apparent. For example, even if as few as 552 common terms are stored as bitvectors rather than as byte coded lists (denoted in the graph as hyb,16), on average half of the query terms in queries of five or more words are stored as bitvectors, and handled more efficiently than would be the case using a pure svs+bc approach. Obviously, the larger the number of common terms stored as bitvectors, the greater the fraction of any particular query that is likely to be able to be handled via bitvector manipulations.

Figures 4 and 5 show measured querying cost across the range of query lengths in the test sequence, using Method One and Method Two processing respectively. In both Figure 4 and Figure 5 the solid black reference line indicates the cost of using a purely bitvector index to process the queries; and the dotted black line shows the speed attained by a binary coded index and svs processing using exponential search to skip forwards through the lists.

Looking at Figure 4 if the index uses bitvector form for terms that satisfy $f_t > n/32$, and a byte coded representation for all other lists, then the Method One hybrid approach is always better than the pure bitvector approach by a slender margin. Note, however, that the query cost does tend to increase as terms are added to the query, and that for long queries an uncompressed svs mechanism can be faster.

Figure 5 then shows the additional usefulness of Method Two processing. Short queries are handled equally quickly as with Method One, and long queries are handled nearly three times faster, within the same amount of index space. In addition, the hybrid bitvector approach, together with Method Two processing, improves completely on the svs+bc+aux,k=4 method described in our previous paper. We also explored combining the auxiliary-indexed byte coded lists with hybrid bitvectors, but got no additional gain. That is, the auxiliary index of Culpepper and Moffat [2007] provides faster (than strictly sequential) processing of the long inverted lists, but storing them as bitvectors is another – apparently even more efficient – way of tapping the same underlying opportunity.

Finally, Figure 6 (using the same 2.8 Ghz Intel Xeon with 2 GB of RAM) revisits the speed/data tradeoff graph that was shown in Figure 1, and illustrates the improvement attained by the hybrid bitvector storage and processing strategies. Method Two, shown in the graph, is faster than Method One, and quite comprehensively outperforms the previous approaches, including the svs+bc+aux auxiliary index method [Culpepper and Moffat, 2007].

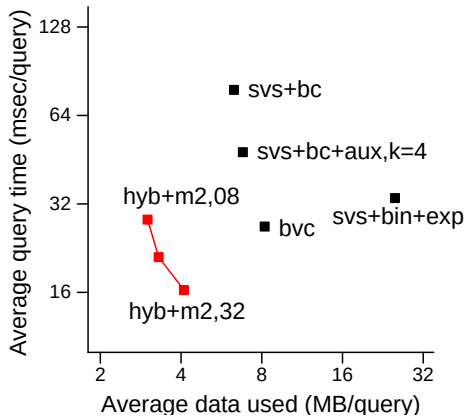

Figure 6: *Space versus CPU cost of different ways of computing conjunctive Boolean queries, showing the gain in speed attained by the hybrid bitvector approach (Method Two). The horizontal axis shows the average per-query data volume required to process a set of 27,004 queries; the vertical axis shows the average time taken over the same query set. Note the shift in both axes compared to Figure 1.*

6 Related work

Previous approaches to providing fast Boolean conjunction in inverted indexes have focused on the provision of internal structures to facilitate pseudo-random access via skipping or similar arrangements [Moffat and Zobel, 1996, Strohman and Croft, 2007, Culpepper and Moffat, 2007]; or on providing compression regimes that allow pointers to be skipped with only partial decompression being necessary [Anh and Moffat, 1998, Scholer et al., 2002, Anh and Moffat, 2006]. Other methods for fast intersection – including adaptive multi-way approaches – have been described in terms of uncompressed binary representations, so that random access operations can be performed [Demaine et al., 2000, 2001, Barbay and Kenyon, 2002, Gupta et al., 2006, Barbay et al., 2006, Sanders and Transier, 2007]. Our work here – which, as noted, builds on an observation made in a previous paper [Culpepper and Moffat, 2007] – is, we believe, the first to fully balance random-access processing of candidate answer documents against frequently occurring terms, with suitably compressed representations for all terms.

7 Conclusion

We have shown that a relatively simple combination of techniques allows fast calculation of Boolean conjunctions within a surprisingly small amount of data transferred. This approach exploits the observation that queries tend to contain common words, and that representing common words via a bitvector allows random access testing of candidates, and, if necessary, fast intersection operations prior to the list of candidates being developed. By using bitvectors for a very small number of terms that (in both documents and in queries) occur frequently, and byte coded inverted lists for the balance,

we have reduced both querying time and also query-time data-transfer volumes.

The techniques described here are not, of course, applicable to other more powerful forms of querying. For example, index structures that support phrase and proximity queries have a much more complex structure, and are not amenable to storage (in their full form) using bitvectors. Nevertheless, there may be scope for evaluation regimes that make use of preliminary conjunctive filtering before a more detailed index is consulted, in which case the structures described here would still be relevant. We plan to explore this option as we continue our investigation into hybrid bitvector structures.

Acknowledgment The query log was supplied by Microsoft Search.

References

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 2006.

V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedingsof the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1998)*, pages 290–297, Melbourne, Australia, August 1998. ACM Press, New York.

J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In D. Eppstein, editor, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 390–399, January 2002.

J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In C. Àlvarez and M. J. Serna, editors, *Experimental Algorithms, 5th International Workshop (WEA 2006)*, volume 4007 of *LNCS*, pages 146–157. Springer, May 2006.

N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, volume 2857 of *LNCS*, pages 122–136, Manaus, Brazil, October 2003. Springer.

J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. P. Consens and G. Navarro, editors, *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE 2005)*, volume 3772 of *LNCS*, pages 1–12, Buenos Aires, Argentina, November 2005. Springer. URL `http://dx.doi.org/10.1007/11575832_1`.

J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In N. Ziviani and R. Baeza-Yates, editors, *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE 2007)*, volume 4726 of *LNCS*, pages 137–148, Santiago, Chile, October 2007. Springer. URL `http://dx.doi.org/10.1007/978-3-540-75530-2_13`.

E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752, January 2000.

E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments (ALENEX 2001)*, volume 2153 of *LNCS*, pages 91–104. Springer, January 2001.

A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In C. Àlvarez and M. J. Serna, editors, *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, volume 4007 of *LNCS*, pages 158–169. Springer, May 2006.

F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered list. *SIAM Journal on Computing*, 1:31–39, 1972.

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

P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 71–83. SIAM, January 2007.

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, and K. Järvelin, editors, *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2002)*, pages 222–229, Tampere, Finland, August 2002. ACM Press, New York.

T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In C. L. A. Clarke, N. Fuhr, N. Kando, W Kraaij, and A. P. de Vries, editors, *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 175–182, Amsterdam, The Netherlands, July 2007. ACM Press, New York.

I. H. Witten, A. Moffat, and T. A. 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. *ACM Computing Surveys*, 38(2):1–56, 2006.