Compact Set Representation for Information Retrieval

J. Shane Culpepper and Alistair Moffat

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

Abstract. Conjunctive Boolean queries are a fundamental operation in web search engines. These queries can be reduced to the problem of intersecting ordered sets of integers, where each set represents the documents containing one of the query terms. But there is tension between the desire to store the lists effectively, in a compressed form, and the desire to carry out intersection operations efficiently, using non-sequential processing modes. In this paper we evaluate intersection algorithms on compressed sets, comparing them to the best non-sequential array-based intersection algorithms. By adding a simple, low-cost, auxiliary index, we show that compressed storage need not hinder efficient and high-speed intersection operations.

1 Introduction

Conjunctive Boolean queries are a fundamental operation in modern search engines. They are used for both traditional AND-mode querying, and also in ranked querying environments when dynamic pruning techniques are used, or when pre-computed static scores such as PageRank contribute to answer ordering [Zobel and Moffat, 2006].

In abstraction, a conjunctive query q is handled by performing a $|q|$-way intersection over $|q|$ ordered sets of integers, with each set being drawn from a pre-computed index and representing the documents containing one of the query terms. In this abstraction, any efficient algorithm can be used to compute the set intersection. But there is considerable tension between the desire to compress the index lists, and the need to process them using efficient intersection algorithms. In particular, the majority of set-versus-set and multi-way merging algorithms that have been described make use of non-sequential access to the elements of the set and are thus at odds with standard sequential decompression methods for compressed data.

In this paper we evaluate intersection algorithms on compressed sets, comparing them to the best array-based intersection algorithms. Because sequential decompression implies linear search, compressed methods seem fated to be slower than array-based alternatives. But, by adding a simple and low-cost auxiliary index, we show that compressed storage need not hinder efficient and high-speed intersection operations.

2 Sets and Set Operations

Techniques for the manipulation of sets and set data have been a rich area of research for several decades. At the most basic level, set manipulation can be reduced to the classical *dictionary problem*, with three key operations needed:

INSERT(S, x)	Return $S \cup x$.
DELETE(S, x)	Return $S - x$.
MEMBER(S, x)	Return TRUE and a pointer to x if $x \in S$; otherwise return FALSE.

Standard efficient structures supporting this group of operations include binary, balanced, and self-adjusting trees; hash tables; and so on. If the *universe* over which the sets are drawn is dense, and readily mapped to the integers $1 \ldots u$, for some value u, then direct-access structures such the *bitvector* can be used, in which the space required is proportional to u rather than to n, the number of elements in the set. Different families of operations may be needed in some applications. For example:

INTERSECT(S, T)	Return $S \cap T$.
JOIN(S, T)	Return $S \cup T$.
DIFFERENCE(S, T)	Return $S - T$.

These high-level set operations are often implemented using a number of more primitive operations. In the next group, there is a notion of "the current element", and as the sequence of operations unfolds, the locus of activity proceeds from one element to another, and the current element migrates around the set:

PREDECESSOR(S)	Return a pointer to the element in S that immediately precedes the current one.
SUCCESSOR(S)	Return a pointer to the element in S that immediately follows the current one.
F-SEARCH(S, x)	Return a pointer to the least element $y \in S$ for which $y \geq x$, where x is greater than the value of the current element.

For example, intersection operations on sets of comparable size can be readily be implemented using the SUCCESSOR operation; and intersection and union operations on sets of differing size can be implemented using the F-SEARCH operation, as described in more detail in Section 3.

The three mid-level set operations can, in turn, be implemented on top of two basic set operations:

RANK(S, x)	Return $\lvert\{y \mid y \in S \text{ and } y \leq x\}\rvert$.
SELECT(S, r)	Return a pointer to the r th largest element in S.

For example, SUCCESSOR(S) can be implemented as SELECT$(S, 1 +$ RANK$(\textit{curr}))$, where *curr* is the value of the current item. In addition, once the RANK and SELECT operations are available, the use of strategically chosen indices in a sequence of SELECT operations can be used to provide an efficient non-sequential implementation of the F-SEARCH operation.

In this paper we are primarily interested in INTERSECT operations, implementing them via a sequence of SUCCESSOR and/or F-SEARCH calls. To set the scene for our evaluation of techniques for implementing these operations, the next section briefly summarizes several key data structures that can be used to represent sets.

3 Set Representations

There are several fundamentally different ways in which sets of integers can be stored, with different attributes, and different ways of handling the basic set operations.

Array of integers: A simple and efficient representation for a set is to store it in a sorted array of integers. For example, a set of $n = 15$ objects over the universe $1 \ldots u$, with (say) $u = 32$, can be stored in 15 words, or, via bit-packing techniques, in 15 five-bit binary numbers. More generally, a set of n items each in the range $1 \ldots u$ can be stored in $n \log u$ bits. Implementation of the operation SELECT in $O(1)$ time is immediate, via array indexing; and MEMBER, RANK, and F-SEARCH require $O(\log n)$ time, $O(\log n)$ time, and $O(\log d)$ time respectively, where d is the number of elements that the current position is shifted by. Large-scale operations such as JOIN and DIFFERENCE take $O(n_2)$ time, where $n_1 \leq n_2$ are the sizes of the two sets involved, because every member of the larger of the two sets might need to be listed in the output array. The final operation, INTERSECT, is considered in more detail in the next section, but can be implemented to require $O(n_1 \log(n_2/n_1))$ time in the worst case.

Bitvectors: Another classic set representation is as a bitvector – a u-bit sequence in which the x th bit is a 1 if and only if $x \in S$. Use of a bitvector shifts the cost balance of the various set operations. All of INSERT, DELETE, and MEMBER take $O(1)$ time; but JOIN, DIFFERENCE, and INTERSECT now take $O(u)$ time, if an output set in the same format is to be constructed. The F-SEARCH, RANK and SELECT operations are also expensive if unadorned bitvector representations are used. But in text querying applications, the output set need not be of the same data type, and can be generated as an array of integers. That means that $O(n_1)$-time intersection is possible via a sequence of MEMBER operations, where n_1 is the size of the smaller set.

A drawback of bitvectors is that their $O(u)$ space requirement is significantly more than the corresponding array representation when $n \ll u$. Also, if the application requires PREDECESSOR and SUCCESSOR query support then the basic bitvector representation may not be efficient.

Jacobson [1988] showed that the addition of a controlled amount of extra space allowed RANK and SELECT to be supported using $O(\log u)$ bit probes, and thus that SUCCESSOR, F-SEARCH and PREDECESSOR could also can be made efficient. Munro [1996] later showed that these operations can be supported in $O(1)$ time. Several further improvements to the original approach have been reported, including some that compress the bitvector by exploiting zones with low "1" densities (for example, [Clark, 1996, Pagh, 2001, Raman et al., 2002]). However, from a practical standpoint, the constant factors to implement the data structures described are high, and these succinct representations are driven by the desire for faster RANK and SELECT operations, neither of which is necessary when processing conjunctive Boolean queries.

Compressed representations: Compact representations of sets are almost all built around a simple transformation, which takes a sorted list of elements and converts them into a set of d-gaps. Any method for coding the gaps as variable length codes, including Huffman codes, Golomb codes, Elias γ and δ codes, and static byte- and nibble-based

codes, can then be used to encode the transformed list. Examples of these approaches are presented by Witten et al. [1999] and Zobel and Moffat [2006].

Given that there are $C_n^u = u!/((u-n)!n!)$ ways of extracting an n-subset from a universe of u possibilities, compressed representations have as their target a cost of

$$\log C_n^u = \log \frac{u!}{(u-n)!n!} \approx n\left(\log \frac{u}{n} + 1.44\right)$$

bits when $n \ll u$. Several of the codes listed in the previous paragraph attain, or come close to, this bound. It is also possible to outperform this worst-case bound if there is significant *clustering* within the set, and the n elements in the set are not a random subset of the available universe. One such code is the Interpolative code of Moffat and Stuiver [2000], which represents sets using binary codes, but in a non-linear sequence that makes it sensitive, or *adaptive*, to any non-uniformity. Adaptivity is explored further below.

The great drawback of most compressed representations is their inability to efficiently support any of the key set operations other than SUCCESSOR. In particular, none of F-SEARCH, RANK, and SELECT are efficient. Indeed, in the face of sequential compression techniques based on d-gaps, a search that shifts the current element by d positions requires $O(d)$ time. To regain faster F-SEARCH functionality, additional information can be added in to compressed set representation. For example, Moffat and Zobel [1996] explore adding periodic *skip* information into the compressed set, so that forward jumps can be taken. They suggest inserting such synchronization points every $O(\sqrt{n})$ positions, and demonstrate improved performance on conjunctive Boolean queries and pruned ranked queries, compared to sequential decoding.

More recently, Gupta et al. [2006] describe a two-level structure in which each of the levels is itself a searchable structure containing compressed information, extending earlier work by Blandford and Blelloch [2004]. In the Gupta et al. method, each block of elements at the lower level is a compressed sequential representation of a balanced binary search tree, stored using a pre-order traversal, and with a skip pointer inserted after each node so that its left subtree can be bypassed if the search is to proceed next into the right subtree. Sitting on top of these blocks is a further data structure that allows the correct tree to be quickly identified. By balancing the sizes and performance of the two structures, good performance is achieved.

4 Intersection Algorithms

This section reviews methods for intersecting sets in array-based implementations, before considering the cost of intersecting compressed sets.

Intersecting two sets: There is an interesting duality between set intersection techniques and integer compression methods. To explore that duality, consider the standard paradigm for calculating the intersection of two sets that is shown in Algorithm 1, where $n_1 = |S| \le n_2 = |T|$, and each of the elements of the smaller set S is searched for in turn the larger set, with the search always moving forward.

Assuming an array representation of T, there are a range of options for implementing F-SEARCH. One is to use a full binary search, taking $O(\log n_2)$ time per operation,

Algorithm 1 Standard two-set intersection, INTERSECT(S, T).

```
1: without loss of generality assume that n_1 = |S| ≤ n_2 = |T|
2: set A ← { }
3: set x ← FIRST(S)
4: while x is defined do
5:     set y ← F-SEARCH(T, x)
6:     if x = y then
7:         add x to A
8:     set x ← SUCCESSOR(S)
```

and $O(n_1 \log n_2)$ time overall. This approach is the dual of the $O(n \log u)$ cost of using binary codes to store an n-item set over the universe $1 \ldots u$. Another simple approach is to implement F-SEARCH as a linear search from the current location, so that forwards traversal over d elements, requires $O(d)$ time. This approach is the dual of storing a set using a Unary code, which in turn is equivalent to the use of a bitvector.

Better algorithms for array-based intersection also have dual codes. The Hwang and Lin [1973] intersection approach corresponds to the use of a Golomb code (see Witten et al. [1999]) to represent a subset. In the Golomb code, a gap of $d \geq 1$ is represented by coding $1 + (d-1)$ div b in Unary, and then $1 + (d-1)$ mod b in Binary, choosing parameter b as $(\ln 2) \cdot (u/n) \approx 0.69(u/n)$. Similarly, in the Hwang and Lin intersection algorithm a parameter $b = 0.69((n1 + n2)/n_1)$ is computed, and the F-SEARCH operations in Algorithm 1 are implemented by stepping b items at a time from the current location in T, and reverting to a binary search over a range of b once a straddling interval has been determined. When Algorithm 1 is coupled with this Golomb-Hwang-Lin searching method, the time required is $O(n_1 \log(n_2/n_1))$, which is worst-case optimal in an information-theoretic sense.

Other integer codes also have duals. The Elias γ code (see Witten et al. [1999]) is the dual of the exponential search mechanism of Bentley and Yao [1976] (referred to by some authors as "galloping" search), and has also been used as a basis for F-SEARCH operations. In the Elias γ code, the representation for a gap of d requires $1 + 2\lfloor \log d \rfloor = O(\log d)$ bits, and a subset of n of u elements requires at most $n\,(2 \log(u/n) + 1)$ bits. Similarly, the Baeza-Yates [2004] non-sequential intersection algorithm is the dual of the Interpolative code of Moffat and Stuiver [2000], mentioned above. In this method, the median of the smaller set is located in the larger set. Both sets are then partitioned, and two recursive subproblems handled.

Adaptive algorithms: There has also been interest in the *adaptive* complexity of codes, and hence (in the dual) of adaptive intersection methods. For example, if all elements in S happen to be smaller than the first element in T, then an implementation of F-SEARCH using linear search, or an exponential search, will execute in $O(n_1)$ time. On the other hand the Golomb-based searching approach is non-adaptive and still gives rise to an $O(n_1 \log(n_2/n_1))$ execution time, even for highly favorable arrangements.

Intersecting multiple sets: When there are multiple sets to be intersected, as is the situation in a text retrieval system handling multi-word queries, the operations can either be implemented as a sequence of binary set intersections, or as a single operation on

Algorithm 2 : The Max Successor intersection algorithm.

```
1: without loss of generality assume that |S_1| ≤ |S_2| ≤ ··· ≤ |S_|q||
2: set A ← { }
3: set x ← FIRST(S_1)
4: while x is defined do
5:    for i = 1 to |q| do
6:       set y ← F-SEARCH(S_i, x)
7:       if x ≠ y then
8:          set x ← max(y, SUCCESSOR(S_1))
9:          break
10:      else if i = |q| then
11:         add x to A
12:         set x ← SUCCESSOR(S_1)
```

multiple sets. Both approaches have their advantages and disadvantages. Set versus set methods (svs) start with the smallest set, and in turn intersect it against each of the others, in increasing order of size. Because the pivotal set of candidates can only get smaller, the worst-case cost of this approach in an array-based implementation is

$$\sum_{i=2}^{|q|} n_1 \log \frac{n_i}{n_1} \le n_1(|q|-1) \log \frac{n_{|q|}}{n_1} \,,$$

where the ordering on the sets is such that $n_1 \le n_2 \le \cdots \le n_{|q|}$. This method is both simple to implement, and also localized in its data access pattern – only two sets are in action at any given time. Either the standard intersection approach shown in Algorithm 1 can be used, or the adaptive Baeza-Yates method can be used.

The other approach is to process all sets simultaneously, and determine the elements in their intersection in an interleaved manner. The simplest approach is to take each element of the smallest set in turn, using it as an *eliminator*, and search for it in the other sets until either it is found in all of them, or is not found in one of them. If it is found, it is then part of the answer; if it is not found in one of the tests, it is eliminated, and the next item from the smallest set is taken.

Demaine et al. [2000] suggested that the set ordering should be dynamic, and be based at all times on the number of remaining elements, so that the cost of every operation is minimized in a greedy sense. As an alternative to the meta-cost of keeping the collection of sets ordered by their number of unprocessed values, Barbay and Kenyon [2002] suggested that the eliminator be chosen instead from the set that caused the previous eliminator to be rejected, so that all sets have the opportunity to provide the eliminator if they are the cause of a big "jump" in the locus of activity. Both of these modified approaches – referred to as adp and seq respectively in the results that appear below – are only of benefit if the input sets are non-uniform with respect to the universe. Barbay et al. [2006] provide a useful overview of how the different search and intersection techniques interact, and summarize a range of previous work.

Max Successor: We also tested another method, max. It takes eliminators from the smallest set, but when the eliminator is beaten, takes as the next eliminator the larger of

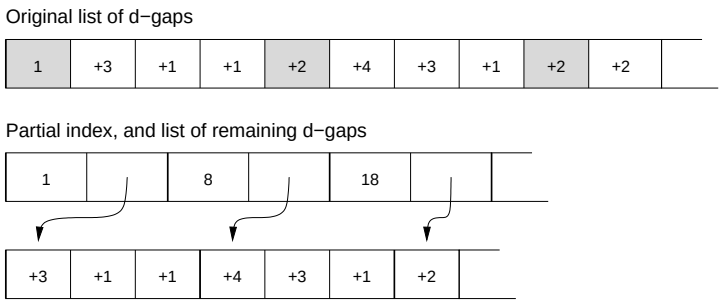

Fig. 1. Extracting every pth document number, and storing it in full in an auxiliary array. In this example $p = 4$, so every fourth d-gap from the list of d-gaps (top row, shaded entries) is extracted and expanded and stored in the auxiliary array (middle row), together with a byte offset into each block of $p - 1$ remaining d-gaps (bottom row).

the item that beat it, or the successor from the first list. Processing then starts at the first list again. Algorithm 2 describes this new approach.

5 Practical indexing

To provide practical access to compressed sets, we return to a classic algorithmic theme, and build a partial index into the list of compressed d-gaps. Figure 1 sketches the proposed arrangement. Every pth d-gap is removed from the compressed index list, expanded into a document number, and stored in the auxiliary index. A bit offset (or byte offset, for byte-aligned codes) is also stored, as shown in the middle row in Figure 1. To search for a value x, the auxiliary index is first searched, to determine a containing block. Any of the available searching methods can be employed. Once the block that might contain x is identified, it is sequentially decoded and d-gaps resolved, starting at the access pointer. The cost of searching a set of n values is thus at most $O(\log(n/p))$ values accessed for a binary search in the auxiliary index, plus $O(p)$ values decoded during the linear search within the block. Taking $p = k \log n$ for some constant k gives search costs that are $O(\log n)$.

To compute the storage cost of the altered arrangement, suppose that the underlying compression method is an efficient one, and that the full set of n original d-gaps is stored in $n \log(u/n)$ bits. Removing every pth gap multiplies that by $(p-1)/p$. Each of the n/p entries in the auxiliary index requires $\log u$ bits for an uncompressed document number, and $\log(n \log(u/n)) \le \log n + \log\log u$ bits for the access pointer, totaling

$$\frac{p-1}{p} n \log \frac{u}{n} + \frac{n}{p} \left(\log u + \log n + \log\log u\right)$$

bits. If we again take $p = k \log n$, this simplifies to

$$n \log \frac{u}{n} + \frac{n}{k} \left(2 + \frac{\log\log u}{\log n}\right) .$$

When $n \geq \log u$, the overhead cost of the auxiliary index is thus $O(n/k)$ bits, with a search cost of $O(k \log n)$ time. In real terms, when $k = 1$, the cost of the auxiliary index is two bits per pointer in addition to the compressed storage cost of the index lists.

One drawback of this hybrid approach is that F-SEARCH operations over a distance of d are no longer guaranteed to take $O(\log d)$ time. For example, a search operation that shifts the current location forward by $d = \log n$ items in a list containing n pointers must, of necessity, involve sequential decoding within the next block of d-gaps, and thus $O(\log n)$ time. One of the objectives of our experiments was to determine the extent to which this issue affected practical operations.

Compared to the *skipped inverted lists* of Moffat and Zobel [1996], our blocks are much shorter, the auxiliary index is maintained separately to the main sequence of d-gaps rather than interleaved with it, and the auxiliary index is stored uncompressed. These differences add to the space requirement of the inverted index, but allow faster F-SEARCH operations, and thus faster INTERSECT computation. In recent independent work, Sanders and Transier [2007] also investigate two-level representations to improve intersection in compact sets. The main focus of their work is a variation on most significant bit tabling to create buckets of roughly uniform size. Sanders and Transier also consider the possibility of deterministic bucket sizes, in a method similar to the approach proposed here.

6 Experiments

This section describes the arrangements used to measure the execution cost of different set intersection techniques in an environment typical of text search engines.

Collection and queries: All of our results are based on experiments with a large set of queries, and the GOV2 collection of 426 GB of web data used in the TREC Terabyte Track (see `http://trec.nist.gov`). This collection contains just over 25 million documents, and about 87 million unique words. For our measurements, words that appeared only one or twice in the collection were assumed to be handled directly in the vocabulary rather than via index lists, and this meant that a total of 19,783,975 index lists were considered. Each list was an ordered sequence of document numbers in the range 1 to $u = 25{,}205{,}181$. Table 1 lists the cost of storing those lists using different representations. For example, stored as uncompressed 32-bit integers, the index requires 23 GB, compared to a combinatorial set cost, summed over all the lists, of 6 GB. Byte codes do not attain the latter target, nevertheless they provide an attractive space saving compared to uncompressed integers, and an even greater saving compared to bitvectors.

The queries used against this collection were derived from a set supplied by Microsoft as being queries for which at least one of the top three answer documents was in the `.gov` domain, as of early 2005. A total of 27,004 unique multi-word queries in the set had conjunctive Boolean matches in the GOV2 collection. Table 2 shows the distribution of query lengths in the query set, and the range of set sizes involved. Note how, even in two term queries, the most common term appears in more than 5% of the documents. The average query length tested was 2.73 which is near the expected average query length of 2.4 [Spink et al., 2001].

Table 1. Total space cost in gigabytes to store (in the center column) all of the inverted lists for the 426 GB TREC GOV2 collection, and (in the right column) the subset of the inverted lists referred to by the experimental query set.

Data Representation	TREC GOV2 19,783,975 words	Query Set 15,208 words
Bitvector	58,051.4	44.6
Integer (32-bit)	22.6	14.1
d-gaps, byte code, and auxiliary index, $k = 2$	8.5	4.4
d-gaps and byte code	7.4	3.8
Combinatorial cost	5.9	2.7

Table 2. Length distribution of the 27,004 queries. The average query length is 2.73 terms.

query length $\|q\|$	2	3	4	5	6	7	8	9	10+
number of queries	15,517	7,014	2,678	1,002	384	169	94	44	102
matches ('000)	124	78	56	41	27	15	10	11	3
average n_1 ('000)	338	325	348	356	288	248	165	226	112
average $n_{\|q\|}$ ('000)	1,698	5,725	10,311	14,317	15,927	17,365	17,958	18,407	19,236

Measurement: To carry out experiments, the index lists for the 15,208 words that appeared in the query set were extracted from the index into a separate file, as shown in the right-hand column of Table 1. Low frequency terms are rarely queried, and the cost of the bitvector representation drops dramatically. On the other hand, the relative fractions of the compressed representations suggest that more than half of the full index still needs to be manipulated.

The set of queries was then executed using the various intersection algorithms. To process one query, the set of lists pertinent to that query was read into memory while the execution clock was stopped; the clock was then monitored while the query was executed five times in a row to generate a list of answer document numbers in $1 \ldots u$; and then the CPU time taken by the five operations was added to a running total, according to the length of that query. For example, the time recorded for queries of length two is the average of $5 \times 15{,}517 = 77{,}585$ executions of 15,517 different queries. We also recorded the number of comparisons performed by each method, as a check against previous results, but report only CPU times here.

Array-based intersection: The first question was the extent to which the adaptive methods were superior to the standard ones. There are two levels at which adaptivity is possible – by using exponential search rather than the worst-case optimal Golomb search; and by performing all $|q| - 1$ merge operations in tandem with an enhanced choice of eliminator at each step, as described in the previous section. The results of these first experiments are shown in Figure 2, where it is assumed throughout that the sets are stored as arrays. In the left-hand graph, Golomb search is used with three multi-way methods, and the svs approach. The right-hand graph shows the same experiment, but using exponential search. In both arrangements the svs ordering outperforms the

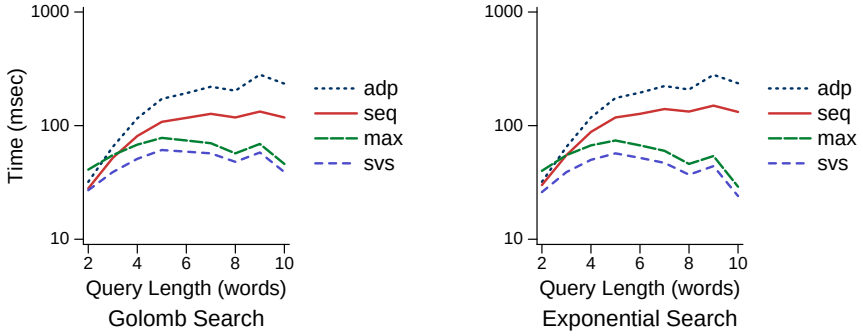

Fig. 2. Efficiency of different intersection algorithms with non-sequential search mechanisms, for different query lengths in the TREC GOV2 dataset, and an array set representation. Methods adp, seq, and max are multi-way methods; svs is the set-versus-set approach.

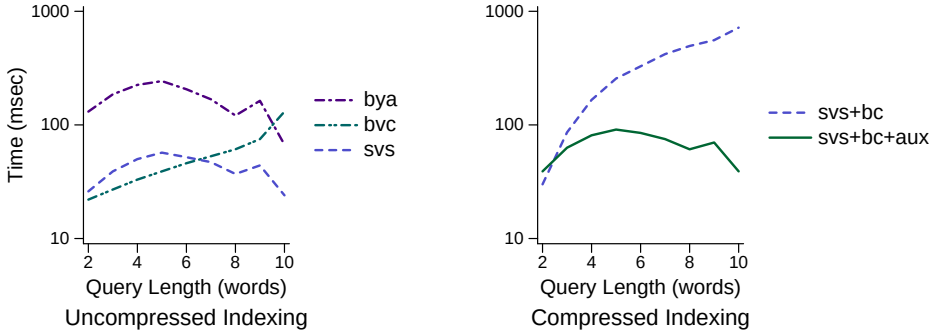

Fig. 3. Efficiency of intersection algorithms for different query lengths in the TREC GOV2 dataset on a 2.8 Ghz Intel Xeon with 2 GB of RAM: (a) fast methods, including the use of random access, where svs is set-vs-set, using exponential search, bya is set-vs-set using the Baeza-Yates method and binary search, and bvc is a bitvector-based evaluation; and (b) two methods that use compressed data formats, where svs+bc involves sequential processing of byte codes, and svs+bc+aux makes use of byte codes indexed by an auxiliary array with $k = 2$.

multi-set approaches, presumably as a consequence of the more tightly localized memory access pattern. Comparing the two graphs, the svs method benefits slightly from exponential search. Note also that execution time tends not to grow as more terms are added to the query – the cost is largely determined by the frequency of the rarest element, and long queries are likely to use at least one low-frequency term.

We also tested the binary search-based Baeza-Yates [2004] method, which is adaptive by virtue of the search sequence. It operates on two sets at a time, but has little locality of reference, and was slower than the svs approach.

Compressed indexing: Figure 3 compares uncompressed representations with two different compressed representations, in all cases using an underlying set-versus-set approach. In the left-hand graph the three best methods are shown – two using array

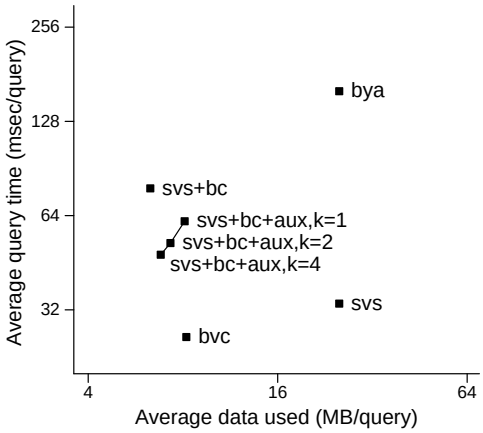

Fig. 4. Tradeoffs between index cost and query throughput, plotted as the average amount of data processed per query over all $27{,}004$ queries, versus the average time taken per query.

representations and non-sequential searching methods, and one (labeled bvc) based on bitvector operations. In the right-hand graph both lines refer to indexes stored in compressed form using byte codes. In the svs+bc approach, F-SEARCH operations are handled via sequential access and linear search; and in the svs+bc+aux method, through the use of an auxiliary index array. Use of the index array greatly increases processing speed on long queries, and allows intersections to be handled in times very close to the uncompressed svs cost in the left-hand graph.

Disk traffic: One aspect of our experiments that is not completely faithful to the operations of an information retrieval system is that we have not measured disk traffic as part of the query cost. Figure 4 shows data transfer volumes plotted against query time, in both cases averaged over the mix of $27{,}004$ queries. The svs+bc+aux methods, using the auxiliary array, require only slightly more disk traffic than the fully-compressed svs+bc approach, and execute in as little as half the time. The indexed compressed methods are slower than the uncompressed svs+exp method, using exponential search, but the latter involves more disk traffic. The bvc bitvector approach also provides a surprising blend of data transfer economy (because most query terms are relatively common in the collection) and query speed (because most queries are short). It may be that hybrid approaches involving some terms stored as bitvectors and some using byte codes are capable of even faster performance, and we plan to explore this option next.

Acknowledgment. The second author was funded by the Australian Research Council, and by the ARC Center for Perceptive and Intelligent Machines in Complex Environments. National ICT Australia (NICTA) is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors, *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, volume 3109 of *LNCS*, pages 400–408. Springer, July 2004.

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, May 2006.

J. Bentley and A. C-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.

D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In J. I. Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 11–19, January 2004.

D. Clark. *Compact PAT trees.* PhD thesis, University of Waterloo, 1996.

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.

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, 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, 1973.

G. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988.

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

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

J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (STACS 1996)*, volume 1180 of *LNCS*, pages 37–42. Springer, December 1996.

R. Pagh. Low redundancy in static dictionaries with constant time query. *SIAM Journal on Computing*, 31(2):353–363, 2001. URL `http://www.brics.dk/~pagh/papers/dict-jour.pdf`.

R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In J. I. Munro, editor, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 233–242. Society for Industrial and Applied Mathematics, January 2002.

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.

A. Spink, D. Wolfram, B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science*, 52(3):226–234, 2001.

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.