Efficient set intersection for inverted indexing

J. SHANE CULPEPPER
RMIT University and The University of Melbourne
and
ALISTAIR MOFFAT
The University of Melbourne

Conjunctive Boolean queries are a key component of modern information retrieval systems, especially when web-scale repositories are being searched. A conjunctive query q is equivalent to a $|q|$-way intersection over ordered sets of integers, where each set represents the documents containing one of the terms, and each integer in each set is an ordinal document identifier. As is the case with many computing applications, there is tension between the way in which the data is represented, and the ways in which it is to be manipulated. In particular, the sets representing index data for typical document collections are highly compressible, but are processed using random access techniques, meaning that methods for carrying out set intersections must be alert to issues to do with access patterns and data representation. Our purpose in this paper is to explore these tradeoffs, by investigating intersection techniques that make use of both uncompressed "integer" representations, as well as compressed arrangements. We also propose a simple hybrid method that provides both compact storage, and also faster intersection computations for conjunctive querying than is possible even with uncompressed representations.

Categories and Subject Descriptors: E.2 [**Data Storage Representations**]: Composite structures; H.3.2 [**Information Storage**]: File organization; H.3.3 [**Information Search and Retrieval**]: Search process

Additional Key Words and Phrases: Compact data structures, information retrieval, set intersection, set representation, bitvector, byte-code

1. INTRODUCTION

The computation of set intersections is essential in information retrieval systems. The dominant indexing data structure leveraged by search engines is the *inverted index* [Witten et al. 1999; Zobel and Moffat 2006]. In an inverted index, an ordered set of document identifiers referred to as an *inverted list* or *postings list* is stored for each term that appears in the collection, identifying which documents contain the term. These lists are then processed via conjunctive Boolean queries in order to identify the subset of documents which contain all of a given set of search terms – the *conjunction*, or *intersection* of the sets indicated by the lists.

The literature describes a range of techniques for computing set intersections, falling

Authors' addresses: J. Shane Culpepper, School of Computer Science and Information Technology, RMIT University, Victoria 3001, Australia. Alistair Moffat, NICTA Victoria Research Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia. The work described in this paper was presented in preliminary form at the 14th SPIRE Symposium on String Processing and Information Retrieval, Santiago, November 2007; and at the 12th ADCS Australasian Document Computing Symposium, Melbourne, December 2007. National ICT Australia (NICTA) is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

broadly into two groupings:

—approaches that assume that the set are stored as sorted arrays of integers, and thus that the dth integer in the list can be accessed in $\mathcal{O}(1)$ time; and

—approaches that assume that the lists are stored and accessed sequentially, possibly in some compressed form, and thus that accessing the dth integer in the list might take time that grows as a function of d, even if it is sublinear in d.

Our presentation in this paper first explores the execution and cost of methods based around these two quite different starting points, and evaluates their relative performance, assuming throughout that all of the required data is being held in random-access memory. We then re-visit a third set representation, one that is usually dismissed as being inordinately expensive for storing index data, but one that with judicious balancing of concerns provides a notable speed up of set intersection operations, without adding greatly to the overall cost of storing the index. We limit our investigation to in-memory indexes in order to provide a clean abstraction for our efficiency evaluation in a manner similar to other recent empirical investigations [Strohman and Croft 2007; Transier and Sanders 2008]. The next section lays the foundation for all three of these approaches, by describing the underlying operations required of all set data structures.

2. SET OPERATIONS AND SET INTERSECTION

There are two broad categories of operations on sets. Operations which return information derived from the current state of a set are referred to as *query operations*, and operations which change the contents or state of a set are *update operations*.

2.1 Dictionaries and Sets

The *dynamic dictionary* abstract data type assumes that two update operations and one query operation must be supported:

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

The simpler *static dictionary* does not require the two update operations, but does require that a suitable initialization process be provided that creates a queryable structure from a list of the set's members. Such "bulk load" processes are usually more efficient than a sequence of **INSERT** operations would be.

When the objects being stored in a static dictionary are integers, one simple representation of a static dictionary is as a sorted array of explicit values, referred to in this work as the **SAEV** set representation, where "explicit" means that each ordinal integer is stored in unmodified form, independently of the others in the set. The **MEMBER** operation over a set of n items stored in **SAEV** format can be implemented in $\mathcal{O}(\log n)$ time using binary search.

Dictionaries are often used to build more elaborate data structures which support composite set operations that typically manipulate multiple elements in a single atomic operation, and are constructed using one or more of the elemental operators listed above:

INTERSECT(S, T)	Return the set $S \cap T$.
UNION(S, T)	Return the set $S \cup T$.
DIFF(S, T)	Return the set $S - T$.
EQUAL(S, T)	Return **TRUE** if $S = T$, otherwise return **FALSE**.
SPLIT(S, x)	Return the two sets $\{z \mid z \in S \text{ and } z \leq x\}$ and $\{z \mid z \in S \text{ and } z > x\}$.
RANGE(S, x, y)	Return the set $\{z \mid z \in S \text{ and } x \leq z < y\}$.

Note that the **SPLIT** and **RANGE** operations assume that the set is ordered, and that comparisons may be performed on the items being manipulated. If an ordering requirement is added, several more primitive operations can be considered, including a "finger search" mechanism:

PRED(S)	Return a pointer to the element in S that immediately precedes the current one.
SUCC(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 $z \in S$ for which $z \geq x$, where x is greater than the value of the current element.
RANK(S, x)	Return $\lvert\{z \mid z \in S \text{ and } z \leq x\}\rvert$.
SELECT(S, r)	Return a pointer to the r th largest element in S.

In inverted index processing, the **INTERSECT**, **UNION**, and **DIFF** operations can be implemented using **F-SEARCH**, **PRED**, and **SUCC** operations. The latter three operators are *state-modifying*, in that they require that a "current" element have been determined by a previous operation, and in turn move that designator to a different element as a side effect of their execution. As the sequence of operations unfolds, the locus of activity shifts through the set being processed. In some representations, these operations can in turn be built on top of **RANK** and **SELECT**. For instance, **SUCC**(S) can be implemented as **SELECT**$(S, 1 + \textbf{RANK}(S, c))$, where c is the current item.

2.2 Binary Intersection of Ordered Sets

The **INTERSECT** operation is the critical one involved in conjunctive Boolean query processing; and also in ranked query processing when the ranking is a static one, based on fixed attributes of the page, with all presented answers required to contain every query term [Zobel and Moffat 2006]. To resolve these queries, the document collection is preprocessed to generate a set of inverted lists, in which each term is represented by an ordered set of ordinal document numbers in which that term appears. To process a conjunctive Boolean query q containing $|q|$ terms, a $|q|$-way intersection of $|q|$ pre-computed sets is then required.

The simplest case is when $|q| = 2$, and two sets are to be intersected; and the most obvious approach is to spend $\mathcal{O}(|S| + |T|)$ time on a standard sequential merge, picking out the elements in common to the two sets via a loop in which a single comparison is made at each iteration, and depending on the outcome of the comparison, a **SUCC** operation is applied to one or the other of the two lists. This approach is the correct one for **UNION** operations, in which the size of the output list is $\mathcal{O}(|S|+|T|)$ and it is assumed that the two

Algorithm 1 Binary set intersection

INPUT: Two ordered sets S and T, with $|S| = n_1$ and $|T| = n_2$, and $n_1 \leq n_2$.

OUTPUT: An ordered set of answers A.

```
1: A ← { }
2: x ← FIRST(S)
3: while x is defined do
4:    y ← F-SEARCH(T, x)
5:    if x = y then
6:       APPEND(A, x)
7:    end if
8:    x ← SUCC(S)
9: end while
10: return A
```

input sets may not be destroyed during the operation, but is not efficient for **INTERSECT** in the cases when $|S| \ll |T|$.

Algorithm 1 describes a more complex but also more efficient intersection algorithm, in which each element of the smaller set, S, is tested against the larger set, T, and retained if it is present [Hwang and Lin 1972]. The search retains state as it proceeds, with the *eliminator* element, x, stepped through the elements of S; and the **F-SEARCH** (finger search) operation used in T to leapfrog over whole subsequences, pausing only at one corresponding value in T for each item in S. An auxiliary operation, **FIRST**, is used to establish an initial state in S; and T is implicitly assumed to have also been initialized, so that the first **F-SEARCH** starts from its least item.

The essential primitive operations in Algorithm 1 are **SUCC** and **F-SEARCH**, with $|S|$ of each performed. In fact, the **SUCC** operation in set S can be replaced by a symmetric call to **F-SEARCH**(S, y) if **SUCC** is not available as an operation, but the expected number of elements jumped is just one, and in practice **SUCC** is likely to execute faster than **F-SEARCH**.

A range of ways in which **F-SEARCH** can be implemented is discussed in the next subsection, and **SUCC** can for most purposes be assumed to require $\mathcal{O}(1)$ time.

2.3 Algorithms for Efficient F-Search

Binary search over n_2 elements requires $1 + \lfloor \log n_2 \rfloor$ comparisons, and if set T is stored as a sorted array of explicit values (**SAEV** format), then binary search can be used to underpin the **F-SEARCH** operations required in Algorithm 1. In particular, binary search is the optimal approach when $|S| = 1$. As a slight improvement, the current element in T can be used to delimit the search, to gain an incremental benefit on the second and subsequent **F-SEARCH** calls.

There are also other searching methods that can be applied to **SAEV** representations, including linear search, interpolation search, Fibonacci search, exponential search (also referred to as *galloping* search by some authors), and Golomb search [Hwang and Lin 1972]. The desirable characteristic shared by these alternatives is that the search cost grows as a function of the distance traversed, rather than the size of the array. For example, linear search requires $\mathcal{O}(d)$ time to move the finger by d items; and as is described shortly, exponential search requires $\mathcal{O}(\log d)$ time [Bentley and Yao 1976]. A sequential linear

Algorithm 2 Golomb **F-SEARCH**

INPUT: A sorted list L of n elements, a pre-calculated Golomb parameter b, a search key x, and a current position in L indicated by *curr*.

OUTPUT: An offset in the sorted list L if x is found, the offset of the successor of x if not found, or **ENDOFLIST** if x is greater than $L[n]$.

1: *pos* $\leftarrow$ *curr* $+ b$
2: **while** *pos* $< n$ **and** $L[pos] < x$ **do**
3: *curr* $\leftarrow$ *pos*
4: *pos* $\leftarrow$ *curr* $+ b$
5: **end while**
6: **if** *pos* $> n$ **then**
7: *pos* $\leftarrow n$
8: **end if**
9: *offset* $\leftarrow$ **BINARY-SEARCH**$(L[curr + 1 \ldots pos], pos - curr, x)$
10: **if** *offset* $=$ **ENDOFLIST then**
11: *curr* $\leftarrow$ *pos*
12: **else**
13: *curr* $\leftarrow$ *pos* $+$ *offset*
14: **end if**
15: **if** *curr* $> n$ **then**
16: **return ENDOFLIST**
17: **else**
18: **return** *curr*
19: **end if**

merge – dismissed above as being inefficient when $n_1 \ll n_2$ – results if linear search is used in Algorithm 1.

In situations when $1 \ll n_1 \ll n_2$, use of exponential search in the **F-SEARCH** implementation is of considerable benefit. In an exponential search, probes into T are made at exponentially increasing rank distance from the current location, until a value greater than the search key is encountered. A binary search is then carried out within the identified subrange, with this "halving" phase having the same cost as the "doubling" phase that preceded it. In this approach each **F-SEARCH** call requires $1 + 2\lfloor \log d \rfloor$ comparisons, where d is the difference between the rank of the finger's previous position and the new rank of the finger pointer. Over n_1 calls for which $\sum_{i=1}^{n_1} d_i \leq n_2$, the convex nature of the log function means that at most $\mathcal{O}(n_1 + n_1 \log(n_2/n_1))$ comparisons are required. Note that this approach has the same worst case asymptotic cost as using binary search when n_1 is $\mathcal{O}(1)$, and has the same worst case asymptotic cost as linear search when n_2/n_1 is $\mathcal{O}(1)$.

If average comparison cost is of interest, the **F-SEARCH** can make use of the *interpolation search* mechanism [Gonnet et al. 1980]. This searching algorithm has an average execution cost of $\mathcal{O}(\log \log n)$ when the data is drawn from a uniform distribution, but in the worst case might require n comparisons. In practice, the running time of interpolation search is often worse than binary search, because each comparison involves more associated computation and hence more time. However, modern processors are closing the gap between interpolation and binary search, and complex calculations are becoming cheaper than navigating across cache lines [Hennessy and Patterson 2006].

Algorithm 3 Small versus small intersection, svs

INPUT: A list of $|q|$ ordered sets $S_1 \ldots S_{|q|}$. The function **INTERSECT** is defined in Algorithm 1.

OUTPUT: An ordered set of answers A.

```
1: without loss of generality assume that |S_1| ≤ |S_2| ≤ ··· ≤ |S_|q||
2: A ← S_1
3: for i = 2 to |q| do
4:     A ← INTERSECT(A, S_i)
5: end for
6: return A
```

The **F-SEARCH** algorithm can also be based on *Golomb* searching, in a mechanism described by Hwang and Lin [1972]. Algorithm 2 shows an implementation of this technique. Search proceeds in a manner somewhat similar to exponential **F-SEARCH**, but with a fixed forwards step of b items used at each iteration. Once overshoot has been achieved, a binary search takes place over the (at most) b items that have been identified. When searching through a set of size n_2 for the elements of a set of size n_1, the correct value for the step b is $0.69(n_2/n_1)$, with a total search cost that is again proportional to $\mathcal{O}(n_1 + n_1 \log(n_2/n_1))$ [Gallager and van Voorhis 1975].

Variable-length integer coding techniques and **F-SEARCH** algorithms are duals of each other. For example, linear search is the dual of the unary code; binary search the dual of the binary code; the exponential search of Bentley and Yao [1976] is the dual of the Elias C_γ code [Elias 1975]; and the search described by Hwang and Lin [1972] is the dual of the Golomb code [Golomb 1966]. Likewise, the search method of Baeza-Yates [2004] is the dual of the interpolative codes of Moffat and Stuiver [2000], and is conceptually similar to the divide and conquer merging techniques explored by Moffat and Port [1990]. Via this duality it is possible for any integer coding mechanism to be applied to the **F-SEARCH** task and then employed in an intersection algorithm on **SAEV**-representations.

3. MULTI-SET INTERSECTION

When more than two sets are being intersected, the simplest approach is to iteratively apply the standard two-set intersection method using as a sequence of pairwise operations. Algorithm 3 shows this *small versus small* (svs) approach. The smallest set is identified, and then that set is intersected with each of the others, in increasing order of size. The candidate set is never larger than S_1 was initially, so the worst-case cost of this approach using a **SAEV** data representation using an **F-SEARCH** that takes $\mathcal{O}(\log d)$ time to process a jump of length d is given by

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

where it is assumed that the sets are ordered by size, with $n_1 \le n_2 \le \cdots \le n_{|q|}$. The svs method is simple and effective, and benefits from the spatial locality inherent from processing the sets two at a time. Even so, each different **F-SEARCH** implementation gives rise to a different svs computation.

Other svs-based approaches are also possible. For example, the divide and conquer

binary intersection approach described by Baeza-Yates [2004] can also be iterated in a small-versus-small manner.

3.1 Holistic Intersection Algorithms

The alternative to the svs approach is to combine all of the sets using a single concerted sweep through them all. The resultant holistic algorithms offer the possibility of being adaptive to the particular data arrangement present, and can potentially outperform the svs approaches. Still working with the **SAEV** representation, the simplest holistic approach is to treat each item in the smallest set as an eliminator, and search for it in each of the remaining sets. Conceptually, this method is identical to an interleaved version of svs. Other adaptive approaches have been proposed which primarily differ in the way that the eliminators are selected at each iteration. Barbay et al. [2006] provide a detailed overview of how such combinations interact, and summarize a range of previous work.

The two prevailing eliminator selection techniques are the *adaptive* algorithm of Demaine et al. [2000], denoted adp for our purposes; and the *sequential* algorithm (seq) of Barbay and Kenyon [2002]. In adp, the sets are initially monotonically increasing in size. At each iteration, the eliminator is the next remaining item from the set with the fewest remaining elements. If a mismatch occurs before all $|q|$ sets have been examined, the sets are reordered based on the number of unexamined items remaining in each set, and the successor from the smallest remaining subset becomes the new eliminator. This approach reduces the number of item-to-item comparisons expected to be required, but at the possibly non-trivial cost of reordering the $|q|$ lists at each iteration of the main loop.

Barbay and Kenyon [2002] proposed an alternative modification, and suggest that every list should be allowed to supply eliminators. Their *sequential* algorithm, denoted here as seq, uses as the next eliminator the element that caused the previous eliminator to be discarded, and continues the strict rotation among the sets from that point. Only when an eliminator value is found in all the sets – and hence is part of the intersection's output – is a new eliminator chosen from the smallest set. This approach has the advantage that the sets do not need to be reordered, while still allowing all of the sets to provide eliminators. However, this method suffers from a practical disadvantage: more **F-SEARCH** operations are likely to accrue when the eliminator is drawn from a populous set than when it is drawn from one of the sparse sets in the intersection.

3.2 Locality-Dependent, Adaptive Intersection

Holistic methods may have a memory access pattern that is less localized than do svs methods, because all of the sets are processed concurrently. To ameliorate this risk, we propose a further alternative, described by Algorithm 4. The eliminator is initially drawn from the smallest set. When a mismatch occurs, the next eliminator is the larger of the mismatched value and the successor from the smallest set. Processing starts in S_2 if the eliminator is again taken from S_1, otherwise processing begins in S_1. The intuition behind this approach is two-fold. The first is that, while it is true that in the absence of other information, the best eliminator will arise in the smallest set, the likelihood of another set becoming significantly smaller than S_1 during processing is small. The second intuition is that, having discovered a bigger than anticipated jump in one of the sets, that value should naturally be tested against the first set, to see if additional items can be discarded.

Algorithm 4 Max successor intersection, max

INPUT: A list of $|q|$ ordered sets $S_1 \ldots S_{|q|}$.

OUTPUT: An ordered set of answers A.

```
1: without loss of generality assume that |S_1| ≤ |S_2| ≤ ··· ≤ |S_|q||
2: A ← { }
3: x ← FIRST(S_1)
4: startat ← 2
5: while x is defined do
6:    for i = startat to |q| do
7:       y ← F-SEARCH(S_i, x)
8:       if y > x then
9:          x ← SUCC(S_1)
10:         if y > x then
11:            startat ← 1
12:            x ← y
13:         else
14:            startat ← 2
15:         end if
16:         break
17:      else if i = |q| then
18:         APPEND(A, x)
19:         x ← SUCC(S_1)
20:         startat ← 2
21:      end if
22:   end for
23: end while
24: return A
```

3.3 Multi-Set Intersection in Practice

In this section we empirically compare the intersection methods described in the previous sections, still working exclusively in the framework established by the **SAEV** representation. First, we describe the origins of the data used, and consider the statistical properties of two large query sets which form the basis of the empirical study. We then compare and contrast the various **INTERSECT** and **F-SEARCH** combinations, setting the scene for the introduction of other representations in the Sections 4 and 5.

Our experiments are based on the integer lists that comprise the inverted index of the GOV2 collection of the TREC Terabyte Track, see `http://trec.nist.gov`. The total collection contains just over 25 million crawled web documents, roughly 87 million distinct alphabetic "words", and occupies 426 GB of space. The vocabulary and inverted lists were constructed using the `Zettair` search engine, see `http://www.seg.rmit.edu.au/zettair`. Words that appeared with frequency one or two were assumed to be handled outside of the set of inverted lists, for example, within the vocabulary. After this reduction, a total of 19,783,975 index lists remained, each of which was then stored in **SAEV** format as an ordered sequence of document numbers in the range 1 to $u = 25{,}205{,}181$.

Two query sets were used for the experiments. The first query set was extracted from a

Table I. The 27,004 Microsoft queries and 131,433 TREC queries. The average query length for the Microsoft and TREC queries is 2.73 and 4.00 terms respectively. The "average n_i" columns show the average number of integers per index list across all terms for queries of that length.

Query length	Microsoft queries			TREC queries		
	Total	Matches	Average n_i	Total	Matches	Average n_i
$\|q\|$	queries	('000)	('000)	queries	('000)	('000)
2	15,517	124	1,018	28,174	34	560
3	7,014	78	2,433	33,505	21	1,603
4	2,678	56	3,712	28,937	14	2,670
5	1,002	41	4,712	18,864	9	3,445
6	384	27	5,169	10,317	7	4,093
7	169	15	5,746	5,252	6	4,607
8	94	10	6,050	2,772	5	4,999
9+	146	5	5,985	3,612	3	5,502

query log supplied by Microsoft, with the property that every query had a top-three result in the `.gov` domain at the time it was originally executed. The second query set was extracted from the 2005 and 2006 TREC "million-query" track. All single word queries from each set were eliminated, and the remaining queries filtered to ensure that each had at least one conjunctive match in the GOV2 collection. There were a total of $27{,}004$ unique queries retained in the Microsoft query set, all of length two or greater; and $131{,}433$ queries retained in the TREC query set. Table I shows the distribution of query lengths in the two test sequences; the average number of document matches for each query length; and the average number of documents containing each query term, computed for each query as $(\sum_{i=1}^{|q|} n_i)/|q|$. Compared to the Microsoft queries of the same length, the TREC queries tend on average to involve terms that are less frequent in the collection, and thus more selective in terms of answer numbers. Across both logs, the longer queries tend to involve, on average, terms that are more frequent in the collection; but even relatively short queries of three words involved computing intersections over millions of document numbers.

In order to focus solely on the efficiency of the intersection algorithms being tested, the inverted lists for the terms pertinent to each query were read into memory. The execution clock was started, and that query executed five consecutive times, in each case returning the full candidate set of document numbers. The clock was then stopped again, and the CPU time for the run was added to a running total, according to the length of the query evaluated. For example, the time recorded for queries of length two drawn from the Microsoft query set is the average of $5 \times 15{,}517 = 77{,}585$ query executions.

Figure 1 compares the average CPU time per query for each intersection method. The top pair of graphs show the baseline approach of combining binary search with the different **INTERSECT** methods. The second row of graphs similarly shows the performance of the same methods when Golomb **F-SEARCH** is used; the third row, the performance of an exponential **F-SEARCH**; and the final pair of graphs the performance of an **F-SEARCH** based on interpolation search. In each of the four pairs of graphs, the left-hand one shows the measured behavior using the Microsoft query log, and the right-hand one shows the measured behavior using the TREC query log.

Binary search (in the top pair of graphs) is relatively inefficient, as it generates more cache misses than the other methods. The result is a noticeable performance degradation relative to methods which achieve localized access. At the bottom of the figure, interpolation search performs significantly fewer comparisons on average across all **INTERSECT** methods (see Barbay et al. [2006]) and has a lower average case bound, but the added cost

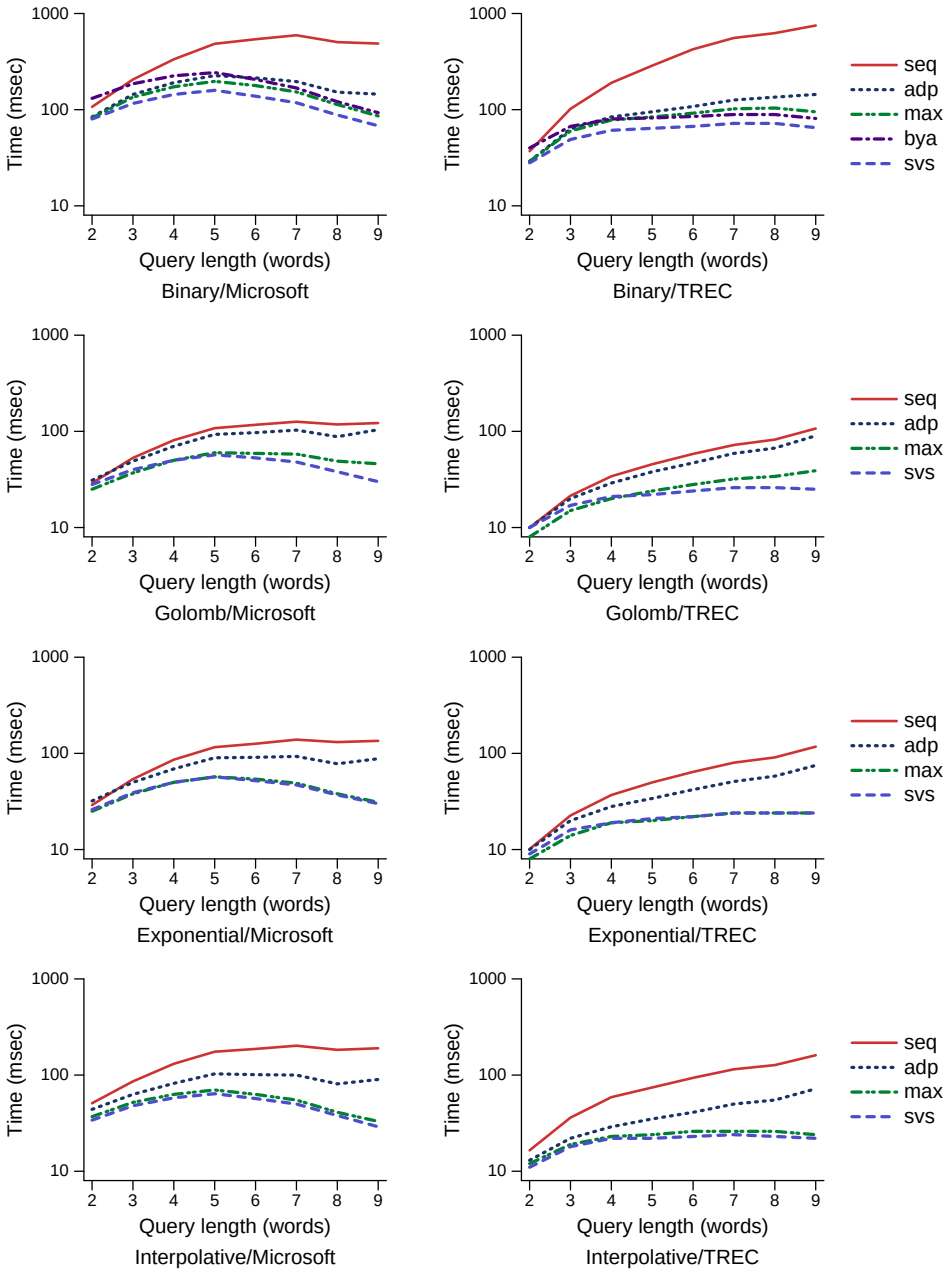

Fig. 1. Execution time required (in CPU milliseconds) of intersection algorithms as a function of query length, for two different query logs processed against the TREC GOV2 dataset, and assuming a **SAEV** set representation. All experiments were carried out on a 2.8 Ghz Intel Xeon with 2 GB of RAM. Note the logarithmic vertical scale.

Table II. Time required (in CPU milliseconds) for the SVS intersection algorithms (which was the most efficient in all cases) as a function of query length and **F-SEARCH** algorithm, for two different query logs processed against the TREC GOV2 dataset, and assuming a **SAEV** set representation. All experiments were carried out on a 2.8 Ghz Intel Xeon with 2 GB of RAM.

Query length	Microsoft queries				TREC queries			
$\|q\|$	Bin	Inter	Exp	Gol	Bin	Inter	Exp	Gol
2	80	34	**26**	28	28	11	**9**	10
3	116	48	**39**	40	49	18	**16**	17
4	144	58	**50**	**50**	61	22	**19**	21
5	159	64	**57**	**57**	64	22	**21**	22
6	138	57	**52**	53	67	23	**22**	24
7	118	50	**47**	48	72	**24**	**24**	26
8	88	38	**37**	38	72	**23**	24	26
9+	68	**29**	30	30	65	**22**	24	25

of the arithmetic involved in calculating each probe means that it is no faster that the other searching approaches. In terms of CPU performance, the svs and max methods outperform the adp and seq multi-set approaches, as a consequence of a more tightly localized memory access pattern.

One issue that arose with the Golomb-based **F-SEARCH** approach depicted in the third row of graphs is the choice of parameter b. In the svs implementation, the value of b appropriate to each list is chosen immediately prior to that list being included in the ongoing sequence of intersections. But with the seq, adp, and max approaches, a value of b is computed for each list at the commencement of processing, based on the length of the shortest list. That same set of b values is then used throughout the computation, which means that if the intersection yields only a few answers, then the searching procedure may use b values that are too low.

The best overall choices across the different eliminator selection approaches are the svs and max mechanisms, coupled with the exponential search or Golomb search for short queries, or coupled with the interpolative search for long ones. When conjunctive queries are computed using these combinations, the execution times of the svs and max methods tend not to grow as more terms are added to the query. This is because the cost is largely determined by the product of the frequency of the rarest element, and the number of terms. The longer the query, the more likely it is to include at least one low-frequency term, with the sum of the marginal savings that accrue on each one of the binary intersections more than recouping the cost of processing a greater number of lists.

Finally, note that the top two graphs also show the binary search-based method of Baeza-Yates [2004], which is adaptive by virtue of the search sequence. It operates on two sets at a time, but within those two sets has little locality of reference when compared to the svs and max approaches, and is a little slower in practice.

Table II shows the average time in milliseconds for each of the four **F-SEARCH** methods when combined with the svs approach, since it always the most efficient in practice. The exponential search (Exp) and Golomb search (Gol) exhibit nearly identical performance across all values of $|q|$, but with Exp having a slight edge throughout the range. This superiority could arise if the eventual answers to each query are not uniformly spread across the document range, and instead tend to form clusters within the document space. Interpolation search (Inter) is also fast when $|q|$ is large, when the last intersections performed involve a very small set. Binary search (Bin) is never the fastest when combined with the svs intersection method. These observations also hold true when these **F-SEARCH**

Table III. The average number of **F-SEARCH** calls initiated for the TREC query set. The average length of the shortest list, n_1, is also shown.

Query length	n_1	svs	adp	bya	seq	max
2	118,138	118,138	118,134	118,139	156,405	**107,191**
3	150,801	183,918	183,086	183,921	273,401	**162,636**
4	161,965	210,971	209,593	210,975	358,891	**184,154**
5	156,616	210,167	208,721	210,172	404,953	**181,733**
6	156,455	213,907	212,114	213,913	454,434	**183,682**
7	154,810	216,606	214,231	216,613	511,181	**186,833**
8	144,740	206,936	205,322	206,943	523,296	**178,123**
9+	127,778	185,787	184,036	185,797	549,895	**160,495**

approaches are coupled with other algorithms, including seq and adp.

Another way of quantifying efficiency is to count the number of **F-SEARCH** calls initiated. Table III shows the average number of **F-SEARCH** calls required across the $131{,}433$ TREC queries, categorized by query length. The number of calls to resolve any given query is largely a function of the query length and the number of terms in the shortest list. However, as queries get longer, on average the rarest query term becomes progressively less frequent, and extended queries can involve fewer **F-SEARCH** calls than shorter ones, provided that an appropriate choice of eliminator is made at each step. In this regard, method seq is clearly more expensive than the others, and the new max method a little better.

From these experiments we conclude that when the sets in question represent term occurrences in documents in a large collection of web pages, and are stored in **SAEV** representation, the classic small-versus-small approach to intersection is the fastest. In combination, the non-parameterized exponential **F-SEARCH** implementation is the approach that is the most versatile across the broad spectrum of query lengths.

4. COMPRESSED SET REPRESENTATIONS

The intersection methods discussed so far assume each set is stored in **SAEV** format as a sorted array of explicit values. This is, however, an expensive way to store the sets that arise from the inverted lists in information retrieval systems, and any claims about relative performance must be re-evaluated when more compact storage representations are employed.

4.1 Compact Sequences of Relative Differences

The index lists in a retrieval system are commonly stored as lists of first-order differences, or *gaps*, between consecutive items [Zobel and Moffat 2006]. Within this broad framework there are then many possible ways of representing the differences, and we will refer to these generically as being *compact sequence of relative differences*, or **CSRD** approaches, and take as axiomatic that the items in the sequence are unique integers and sorted, and thus that the differences are all strictly positive. For example, when converted to differences, the sorted set

$$S = \{1, 4, 5, 6, 8, 12, 15, 16, 18, 20, 25, 26, 27, 28, 30\}\,,$$

is transformed to

$$\{1, 3, 1, 1, 2, 4, 3, 1, 2, 2, 5, 1, 1, 1, 2\}\,,$$

which is then stored using some form of variable-length code that favors small values over large.

As a combinatorial limit, a set of n distinct items drawn from the universe $1 \ldots u$ can be represented in as few as

$$\left\lceil \log_2 \binom{u}{n} \right\rceil = \log \frac{u!}{(u-n)!n!} \approx n \left(\log \frac{u}{n} + 1.44 \right), \tag{1}$$

bits (see Brodnik and Munro [1999] or Sadakane and Grossi [2006] for discussion). Looking at this bound from the point of view of a sequence of n gaps $\langle d_i \mid 1 \le i \le n \rangle$, any coding mechanism that represents the sequence in $\mathcal{O}(n \log(u/n))$ bits, where $u = \sum_{i=1}^{n} d_i$, is a *compact* representation, and is at most a constant factor inefficient compared to the combinatorial lower bound. In particular, any static integer code with the property that the codeword for integer x requires $\mathcal{O}(\log x)$ bits can be used. Codes that meet this requirement include the Elias γ and δ codes [Elias 1975]; and static byte-codes. Golomb codes are explicitly fitted to the situation described, and if all n-subsets of $1 \ldots u$ are equally likely, provide a one-parameter equivalent of the multi-parameter Huffman code that would be derived from the probability distribution governing the d-gaps. Witten et al. [1999] and Moffat and Turpin [2002] describe all of these methods, and we refer the reader to those descriptions rather than repeat them here.

Unfortunately, the **CSRD** representations have a serious drawback: the differencing and compression transformations both produce representations that must necessarily be accessed sequentially. This restriction means that primitive set operations other than **SUCC** are expensive. In particular, the only viable **F-SEARCH** alternative is essentially a linear search, which means that **INTERSECT** of lists of length n_1 and n_2 requires $\mathcal{O}(n_1 + n_2)$ time. Even **MEMBER** queries incur an $\mathcal{O}(n)$ cost when using simple d-gap representations. That is, compact set representations are space effective, but unless they are enhanced with auxiliary structures, are not efficient for set intersection applications such as text retrieval. Recent work on *succinct* set representations which use sophisticated bitvector representations provide a viable alternative to balancing efficiency and effectiveness, but it is unclear how such data structures can be effectively integrated in current inverted index-based systems [Okanohara and Sadakane 2007; Claude and Navarro 2008].

4.2 Bitvectors

A set S of integer values over a known universe $1 \ldots u$ can also be represented using a bitvector, in an approach we refer to as **BITV**. A bitvector is a u-bit sequence in which the xth bit is 1 if and only if $x \in S$. For example, assuming that $u = 32$, the set S in Section 4.1 would be represented as the bitvector $10011101000100110101000011110100$. If the set being represented is dense over the universe $1 \ldots u$, that is, $u/n = \mathcal{O}(1)$, then bitvectors are both space-effective and also access-efficient. On the other hand, when $n \ll u$, bitvectors are more expensive than the compact representations described in the previous subsection. Using a bitvector dramatically shifts the cost balance of several of the primitive set operations. Operations **INSERT**, **DELETE**, and **MEMBER** all take $\mathcal{O}(1)$ time in the **BITV** representation; but **UNION**, **DIFF**, and **INTERSECT** take $\mathcal{O}(u)$ time, and without additional support via auxiliary structures, **F-SEARCH**, **SUCC**, and **PRED** also become more expensive. In practice, **UNION**, **DIFF**, and **INTERSECT** can benefit from *bit-parallelism* to obtain a constant factor speedup, but this does not affect the asymptotic cost of the operations.

Control operations such as **RANK** and **SELECT** are also expensive in unadorned bitvector representations. If the application requires these or other control operations such as

Table IV. Effectiveness of storage, and efficiency of access, for three different representations of integer sets, with emphasis on the operations needed in text retrieval systems, assuming that a subset of n (and a second larger one of size n_2, where the operation has two parameters) items in the range $1 \ldots u$ is being manipulated. In the **F-SEARCH** operation, d is the number of items that are stepped over. In the case of the **BITV** approach and the **F-SEARCH** operation it is assumed that the n elements form a random subset of $1 \ldots u$.

Attribute	**SAEV**	**CSRD**	**BITV**
Space required (bits)	$n \log u$	$n(\log(u/n) + 1.44)$	u
MEMBER (time)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
F-SEARCH (time)	$\mathcal{O}(\log d)$	$\mathcal{O}(d)$	$\mathcal{O}(u/n)$
INTERSECT (time)	$\mathcal{O}(n \log(n_2/n))$	$\mathcal{O}(n + n_2)$	$\mathcal{O}(u)$

Table V. Total space cost (gigabytes) to store three different subsets of the inverted lists of terms appearing in more than two documents in the 426 GB TREC GOV2 collection.

Data Representation	TREC GOV2 19,783,975 words	Microsoft queries 15,208 words	TREC queries 44,862 words
BITV	58,051.4	44.6	131.6
SAEV, 32-bit integers	22.6	14.1	17.1
CSRD, byte codes	7.4	3.8	4.8
Combinatorial limit	5.9	2.7	3.5

F-SEARCH, **PRED**, or **SUCC**, the basic bitvector representation is no longer sufficient. Jacobson [1989] showed that the addition of a controlled amount of extra space allows **RANK** and **SELECT** to be supported in $\mathcal{O}(\log u)$ time. As a consequence, **SUCC**, and **PRED** can also be supported in equivalent time. Building on that work, Munro [1996] demonstrated that **RANK** and **SELECT** can be accomplished in $\mathcal{O}(1)$ time. The new structure, called an "indexable" bitvector, depends on the use of auxiliary lookup tables which store cumulative ranks for blocks of elements. The tradeoff for the performance boost is an additional $o(u)$ bits of space, required to store the lookup tables.

Additional variations on bitvectors have been reported recently, including some which attempt to reduce the space overhead associated with sparse bitvectors (see for example, Clark [1996], Pagh [2001], Raman et al. [2002], Okanohara and Sadakane [2007], and Claude and Navarro [2008]). However, the space overhead can still be significant in certain set arrangements. Note that the crucial motivation for many of the alternative bitvector arrangements is support for efficient **RANK** and **SELECT** operations, neither of which are strictly necessary for set intersection.

Table IV summarizes the three alternative set representations that have been discussed. In each case it is assumed that a subset of n items in the range $1 \ldots u$ is being manipulated, and that the **F-SEARCH** operation steps past d members of the set. The column headed **BITV** further assumes that the set is dense. In the case of the **CSRD** and **BITV** representations, it is also assumed that no auxiliary information is being stored. (But note that in both cases **F-SEARCH** operations can be faster when the base arrangement is augmented by auxiliary index structures.)

4.3 Space Usage in Practice

Table V shows the cost of storing three different subsets of the inverted lists of the GOV2 collection. In the second column, every term that appears in more than two of the collection's documents has its list included in the sum, and the values represent the cost of storing a full index. The third and fourth columns then show the cost of storing inverted lists for

only the subsets of the terms required by the two query sets indicated by the column heading. For example, storing the full set of inverted lists in **SAEV** format using 32-bit integers gives rise to an index of nearly 23 GB. When compared with the combinatorial cost of the full index (as defined in Equation 1, summed over all the lists) of 6 GB, the advantage of storing the index in **CSRD** format is apparent. Byte codes coupled with **CSRD** format do not attain the combinatorial bound, nevertheless they provide attractive space savings compared to uncompressed integers and **SAEV** format. At the other end of the spectrum, storing the full index in **BITV** format is prohibitively expensive; and even if only the required query terms are considered, the **BITV** format – which is the most efficient in terms of **MEMBER** queries – is relatively costly, even if those terms could somehow have been known in advance.

5. BALANCING SPACE AND TIME USAGE

For **CSRD** representations it is possible to accelerate **F-SEARCH** operations by adding additional information to the compressed sequence. For example, Moffat and Zobel [1996] add a *synchronization point* every $\sqrt{n}$ th position, so that blocks of $\sqrt{n}$ items can be bypassed when **MEMBER** and **F-SEARCH** operations involve search keys not present in that block. More elaborate approaches have also been proposed which attempt to offset the costs of a linear scan through all d-gaps using variations on balanced binary search trees. Gupta et al. [2006] describe a two-level data structure where each level is itself searchable and compressed, extending earlier work by Blandford and Blelloch [2004]. In the Gupta et al. [2006] method, each block of elements at the lower level is represented as a balanced binary search tree, and stored implicitly via a pre-order traversal, with additional skip pointers embedded to find each child's left subtree. Sitting on top of each tree block is an index which allows the correct tree to be quickly identified. By balancing the size and efficiency of the two structures, good asymptotic performance is achieved.

5.1 Auxiliary Indexing

In this section we propose a more pragmatic approach to address the same need. The structure, which we refer to as an *auxiliary index*, balances space and time usage in a relatively straightforward manner, and allows fast implementation. It makes use of a partial index of uncompressed items together with offsets into the compressed sequence.

Figure 2 outlines the proposed arrangement. First, every p th element is extracted from the compressed list, added to an auxiliary array, and stored as an uncompressed integer. The choice of a value for p is discussed shortly. Next, the remaining items are represented as d-gaps, and stored as groups of $p-1$ differences, compressed sequentially. Finally, a bit offset (or byte offset, in the case of byte-code compression techniques) is added to the auxiliary array, so that the blocks of the compressed data can be accessed independently of each other.

When a **MEMBER** operation is required, the keys in the auxiliary array are traversed using any **SAEV**-based searching method, including the **F-SEARCH** approaches described in Section 2. If the value being sought is located in the auxiliary array, it is returned. If not, a sequential search is undertaken in the appropriate block of p elements using a **CSRD** mechanism. Alternatively, the block can be completely decompressed into an array, and searched using a **SAEV**-based **MEMBER** operation.

The cost of searching when the set contains n values is at most $\mathcal{O}(\log(n/p))$ for a binary search in the auxiliary index, followed by $\mathcal{O}(p)$-time spent decoding and searching within

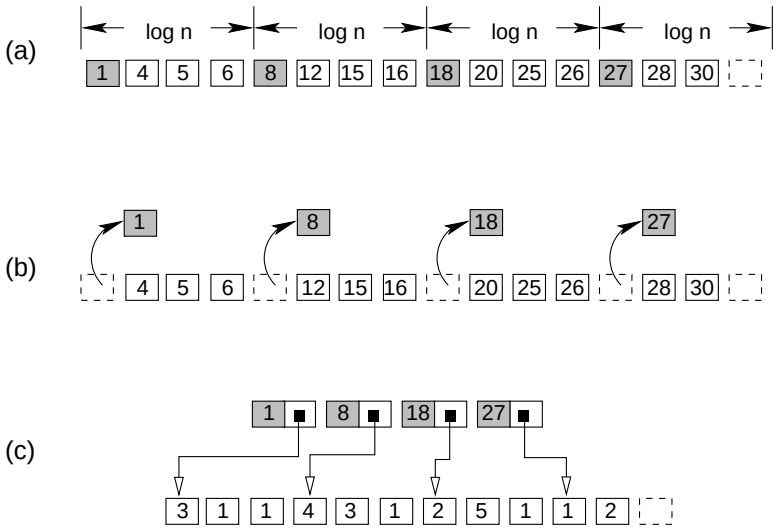

Fig. 2. Auxiliary indexing. In this example $p = 4$, so every fourth element from the original list (part(a), shaded entries) is extracted and stored in the auxiliary array (part (b)), together with a byte offset to the block of $p - 1$ remaining elements. The block elements are then encoded as d-gaps and compressed (part (c)).

the block. If $p = k \log n$, for some constant k, the combined search cost for the two phases is $\mathcal{O}(\log n)$. In terms of space, and assuming an efficient code, the cost of storing each difference is $\log(u/n) + 1.44$ bits in an amortized sense. There are $(p - 1)$ items in each p item block stored in this way, or $(p - 1)n/p$ items in total. The other n/p elements are stored in the auxiliary array, and require $\log u$ bits for each stored element, plus $\log(n \log(u/n)) \le \log n + \log\log u$ bits for the access pointer. Over both components the cost is bounded by

$$\frac{p-1}{p} n \left(\log \frac{u}{n} + 1.44 \right) + \frac{n}{p} \left(\log u + \log n + \log\log u \right)$$

bits, which is readily reformulated as

$$n \left(\log \frac{u}{n} + 1.44 \right) + \frac{n}{p} \left(2 \log n + \log\log u - 1.44 \right) .$$

Then, when $p = k \log n$, this simplifies to

$$n \left(\log \frac{u}{n} + 1.44 \right) + \frac{n}{k} \left(2 + \frac{\log\log u}{\log n} - \frac{1.44}{\log n} \right) ,$$

which is less than

$$n \left(\log \frac{u}{n} + 1.44 \right) + \frac{n}{k} \left(2 + \frac{\log\log u}{\log n} \right) .$$

That is, for an additional $\mathcal{O}(n/k)$ bits compared to Equation 1, the search time in the compact representation reduces to $\mathcal{O}(k \log n)$, provided that $n \ge \log u$. In real terms, if $k = 1$ and $u \le 2^{64}$, the additional cost of the auxiliary index is less than 3 bits per pointer when $n \ge 64$, and less than 2.5 bits per pointer when $n \ge 4096$.

Table VI. Total space cost (gigabytes) of **CSRD** representation and auxiliary indexing, on the same basis as is presented in Table V.

Data Representation	TREC GOV2 19,783,975 words	Microsoft queries 15,208 words	TREC queries 44,862 words
CSRD, byte codes	7.4	3.8	4.8
CSRD, byte codes, auxiliary index, $k = 4$	8.0	4.1	5.1
CSRD, byte codes, auxiliary index, $k = 2$	8.5	4.4	5.5
CSRD, byte codes, auxiliary index, $k = 1$	9.5	4.9	6.2

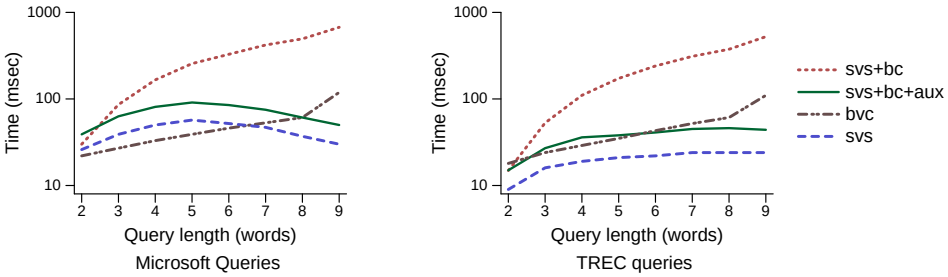

Fig. 3. Performance of set intersection algorithms for different query lengths against the TREC GOV2 dataset, measured using a 2.8 Ghz Intel Xeon with 2 GB of RAM. In each of the two graphs two **CSRD**-based methods are shown: svs+bc, the sequential processing of byte-codes; and svs+bc+aux, byte-codes indexed by an auxiliary array with $k = 2$. Each graph also includes one **SAEV**-based method, svs, a small-versus-small approach using exponential search; and use of a pure bitvector approach, bvc. The vertical scale is the same as was used in Figure 1, and the results here can be directly compared against the two columns of graphs in Figure 1.

A potential disadvantage of the new approach is that **F-SEARCH** operations over a distance of d are no longer guaranteed to take $\mathcal{O}(\log d)$ time. In the worst case, sequential decoding of a block of d-gaps can lead to $\mathcal{O}(\log n)$ time instead. However, this is generally not a significant problem, as the block-based approach is well suited to blockwise hierarchical memory models. When compared with the inline approach described by Moffat and Zobel [1996], our blocks are smaller on average, and the skip pointers are maintained separately from the compressed d-gap sequence. The revised approach allows faster **F-SEARCH** operations, and thus faster **INTERSECT** computations, but at the cost of a small amount of additional space. In recent independent work Sanders and Transier [2007] also investigate two-level representations to improve intersection in compact sets. The key difference between their work and ours is that they use a randomization technique to ensure that each bucket on average has around $\log n$ items, and get the benefit of a simpler auxiliary structure that can be implemented as a look-up table, whereas we count an exact number of elements into each bucket, and hence need to search within the auxiliary table to identify the correct block.

5.2 Auxiliary Indexes in Practice

Table VI shows the total space usage to store the full GOV2 index and two index subsets used in the experiments in **CSRD** format, without the auxiliary index (this row is repeated from Table V), and with an auxiliary index, using three different values of k. The augmented indexes all use more space than the fully byte-coded list, as expected, but even with $k = 1$ the additional cost is relatively modest.

Figure 3 compares the **SAEV** binary representation and the alternative **CSRD** representa-

tion. Fully compressed representations such as the svs+bc approach perform well for short queries, but are much slower than the **SAEV**-based svs approach on longer ones, in which it is more likely that at least some common terms will arise. Use of the auxiliary array in the svs+bc+aux method increases processing speed on long queries, and allows intersections to be handled in times closer to those attained by the uncompressed svs approach. This improvement is a result of intersection benefiting from non-sequential access and, as queries get longer, the increasing sparsity of the most discriminating term list.

Figure 3 also shows the time taken by the bitvector approach, bvc, which is usefully fast on short queries, but as expected, has a cost that grows approximately linearly in the number of query terms. Bitvectors are able to store common query terms in relatively small amounts of space. Indeed, their only real deficiency is the exorbitant storage costs for rare terms, an observation that leads directly to the hybrid approach that is explored in the next section.

5.3 Hybrid Representations

The simplicity of bitvectors, and their useful blend of efficiency and effectiveness for **INTERSECT** and **MEMBER** queries on dense sets, suggests that using them to store at least some of the term lists of the index may be beneficial. To this end, we now consider a hybrid approach, in which the dense lists in the index, corresponding to frequently occurring terms, are stored using a **BITV** representation, and the sparse terms are stored using a **CSRD** representation. Given that each byte-coded difference occupies a minimum of eight bits, using a bitvector for all terms that occur in more than $u/8$ of the documents in a collection containing a total of u documents cannot increase the index size. Other thresholds are also possible, and in the experiments described shortly, the bitvector representation is used whenever more than u/k of the documents contain that term, for values of k in the set $\{8, 16, 32\}$.

Two alternative ways of using the resultant hybrid index have been evaluated. In the first method, denoted hyb+m1, the sets of byte coded and bitvector lists are intersected separately, and then combined using a sequence of $\mathcal{O}(1)$-time **MEMBER** operations. That is, if there are any **BITV** terms, they are intersected using bitwise AND operations to yield a bitvector B. If there are no byte-coded **CSRD** sets, then the expanded set of document numbers derived from B is returned. Or, if there are byte-coded sets, they are intersected using the svs approach to yield a candidate set C. Then, if there are no bitvector sets, C is returned. Otherwise, for each $x \in C$, if $B[x] = 1$, then x is appended to the answer set.

In the second hybrid method, hyb+m2, all byte-coded lists are intersected to produce a candidate list. Then, each of the remaining bitvector lists is searched via a **MEMBER** query, replacing the bitvector **INTERSECT** operations with multiple **MEMBER** operations, each of which takes $\mathcal{O}(1)$ time. The motivation for this alternative comes from the realization that intersection of the byte-coded sets should result in a sparse set of potential candidates, and that a sequence of **MEMBER** queries may be preferable to any $\mathcal{O}(u)$-time **INTERSECT** operations. Algorithm 5 describes this second approach in detail.

5.4 Hybrid Representations in Practice

Table VII shows the cost of storing the GOV2 index data using compressed lists and bitvectors in a hybrid approach. The most effective partitioning point for the compact hybrid representation is $k = 8$, with terms occurring in fewer than $u/8 = 3{,}150{,}648$ documents stored using a **CSRD** approach and the byte-code representation. Note that when $k = 8$

Algorithm 5 Hybrid intersection method two, hyb+m2

INPUT: A list of **CSRD**-representation byte coded term lists $CSRD_1 \ldots CSRD_x$, and a list of **BITV**-representation term lists, $BITV_1 \ldots BITV_y$, where $x + y = |q|$.

OUTPUT: An ordered set of answers, as 32-bit integers.

```
1:  A ← { }
2:  if x = 0 then
3:     B ← BITV_1
4:     for i ← 2 to y do
5:        B ← B bit-and BITV_i
6:     end for
7:     return DECODE-BITV(B)
8:  end if
9:  C ← DECODE-CSRD(CSRD_1)
10: for i ← 2 to x do
11:    C ← INTERSECT(C, DECODE-CSRD(CSRD_i))
12: end for
13: for i ← 1 to y do
14:    for each d ∈ C do
15:       if NOT MEMBER(BITV_i, d) then
16:          DELETE(C, d)
17:       end if
18:    end for
19: end for
20: return C
```

Table VII. Total space cost (gigabytes) of hybrid **BITV**/**CSRD** representations, on the same basis as is presented in Table V and Table VI.

Data Representation	TREC GOV2 19,783,975 words	Microsoft queries 15,208 words	TREC queries 44,862 words
Hybrid bitvector and byte code, $k = 32$	8.9	4.9	6.2
Hybrid bitvector and byte code, $k = 16$	7.3	3.7	4.7
Hybrid bitvector and byte code, $k = 8$	6.9	3.4	4.3

there is a net saving compared to the fully byte-coded index (Table V), because even in the dense term lists, there are some d-gaps that require more than one byte to code. Even with $k = 16$ the overall index is smaller than the byte-coded one, and the same relationship holds for the subsets of the term lists that appear in the two experimental query streams.

Table VIII shows the fraction of the GOV2 collection's index lists that are handled as bitvectors, for three different threshold values k. As a fraction of the vocabulary of the collection, the terms that are sufficiently frequent to warrant bitvector index lists are a tiny minority, even when $k = 32$. But when the two query logs are evaluated, a much greater fraction of bitvector terms arises, because queries tend not to use the rare terms. The top pane of Figure 4 lists the 188 most frequent terms in the GOV2 collection, each of which appears in more than one eighth of the web documents in the collection, so is stored as a bitvector in the hybrid **BITV**/**CSRD** method when $k \geq 8$. The bottom pane then shows a sequence of ten consecutive queries extracted from the TREC query log, with each of

Table VIII. Fraction of index lists stored and processed as bitvectors for three different threshold values k, and two different query streams. Just 188 index lists are stored in **BITV** format when $k = 8$ (listed in Figure 4); even so, one or more of those terms appears in 35% of the Microsoft queries, and in 50% of the TREC queries.

Attribute	Hybrid parameter		
	$k = 8$	$k = 16$	$k = 32$
Fraction of index lists stored in **BITV** format	0.001%	0.003%	0.007%
Fraction of Microsoft queries with one **BITV** index list	24.6%	31.0%	34.6%
Fraction of Microsoft queries with multiple **BITV** index lists	10.7%	21.0%	33.5%
Fraction of TREC queries with one **BITV** index list	28.6%	29.2%	26.8%
Fraction of TREC queries with multiple **BITV** index lists	21.9%	33.7%	47.9%

of to the and for a in on by this is information with home from at or are as not all 1 be us other state search last contact new an page 2 that about national data public may have site if 3 department use you no 2003 available more 4 5 services has privacy center which resources your will it 10 name service number was system help date library can please only also 7 office research 6 one been i these 8 program 2000 15 health related its 12 management through see but links 30 federal states s do report development any index washington support following who year than type 25 when general programs 20 2001 subject first description using where administration out used 2002 their comments modified 11 email policy 23 web 9 questions there time 19 government updated security each map title 16 order make back code find part united law within such accessibility current 14 advanced does 13 both two our how areas years over d they local were c 22 next county includes include 31 disclaimer under agency area into based click need

weather in^{8} london	how^{8} to^{8} find^{8} out^{8} your^{8} body^{32} mas index^{8}
red^{32} dead revolver cheats	grand theft auto san^{32} andreas cheat codes
vacuum tube pins	heaven dj sammy and^{8} yanou
wings of^{8} desire escort ny	vegetable disease^{32} control^{16} sprays
pinnacle village	bike rentals on^{8} the^{8} carolina islands

Fig. 4. The 188 most frequent words in the GOV2 collection, in decreasing order of the number of documents containing them, each of which appears in more than one eighth of the approximately 25 million web documents (top pane); plus a sequence of ten queries extracted from the TREC query set, annotated to show which words are represented as bitvectors for the three different values of k that were tested.

the common terms annotated to show the value of k (of 8, 16, and 32) at which that term toggles from **CSRD** format to **BITV** format in the hybrid arrangement. For example, the query term "control" is stored in **CSRD** format using byte-codes when $k = 8$, but is stored as a bitvector when $k = 16$ and $k = 32$.

Table IX shows the average cost of processing the set intersections arising from the TREC query stream using the hyb+m1 and hyb+m2 methods, broken down by query length, and with techniques built on the **SAEV**, **CSRD**, and **BITV** representations also included. Three different threshold points $k = \{8, 16, 32\}$ are shown. The svs approach, using a **SAEV** data representation, is efficient at all query lengths, and provides a competitive benchmark. Even with the auxiliary index, the **CSRD** data representation is not competitive in terms of query speed, and nor is the pure **BITV** representation. In the latter case, long queries are especially expensive, because of the large amount of index data that must be processed via binary AND operations.

The hyb+m2 mechanism shown in Algorithm 5 is markedly superior to the hyb+m1 approach, with the difference between the two being the elimination of the binary AND operations. Even when the index size is minimized at $k = 8$, queries are resolved in times similar to the svs+bc+aux approach; and when a larger index is permitted using $k = 32$,

Table IX. Time required (in CPU milliseconds) to compute set intersections for the TREC query set against the GOV2 index using a 2.8 Ghz Intel Xeon with 2 GB of RAM, using different set representations, and different intersection techniques. The svs+bc+aux method used a parameter value $k = 4$. When random access operations are available, even long queries can be processed quickly. These times may be directly compared against those shown in the right-hand half of Table II.

$\|q\|$	svs	svs+bc +aux	bvc	hyb+m1			hyb+m2		
				$k=8$	$k=16$	$k=32$	$k=8$	$k=16$	$k=32$
2	9	15	18	10	8	**6**	10	7	**6**
3	16	27	24	19	14	12	19	13	**9**
4	19	36	29	28	22	19	26	17	**11**
5	21	38	35	36	29	26	31	20	**13**
6	22	41	43	45	36	33	36	22	**14**
7	24	45	52	55	44	39	41	24	**15**
8	24	46	61	62	50	45	44	25	**15**
9+	24	44	110	84	65	61	55	28	**17**

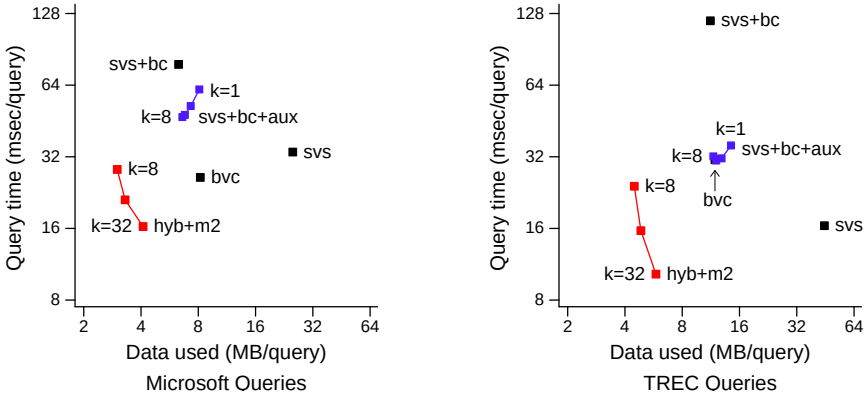

Fig. 5. Tradeoffs between index cost and query throughput, with the average time taken over all queries, in milliseconds per query, plotted as a function of the average amount of index data processed in MB per query, for two different query sequences.

the execution times fall below the benchmark provided by the pure svs approach.

5.5 Bringing It All Together

Figure 5 summarizes the various measurements that have resulted from our experiments. The axes of the two graphs represent the primary resources consumed during querying in a text retrieval system – the average amount of data transferred from secondary storage into memory in order for a query to be resolved (or from memory into cache, in a memory-resident system); and the average CPU time taken per-query to process that data and determine the conjunction of the query terms. In each of the two graphs the average is computed over the full set of queries: 27,004 Microsoft queries in the left-hand graph, and 131,433 TREC queries in the right-hand one. The layout of the graphs means that methods in the top-right corners involve both high data volumes and long query-processing times, and are relatively uninteresting. On the other hand, methods along the lower-left frontier define a range of interesting options.

The "pure" **CSRD** byte-coded index, denoted svs+bc is the slowest of the methods shown. It is not competitive with alternatives that provide fast **MEMBER** queries, and

the simple tactic of adding an auxiliary index (the connected points labeled svs+bc+aux) reduces execution time by a factor of as much as four. The pure **BITV** approach (the point labeled bvc in each graph) is also relatively fast, and requires surprisingly little data transfer – the many low-frequency terms that cause the extreme storage cost of the bitvector index (Table V) are not ones that occur in typical queries; and the terms that do occur in typical queries are relatively efficient when stored as bitvectors. For the same reason, the bvc approach also requires less data to be transferred for average queries than does the pure **SAEV** approach, labeled svs in the graphs, while operating at comparable speed.

The clearly best method amongst those illustrated in Figure 5 is the **BITV**/**CSRD** hybrid method described in Section 5.3. It outperforms the other implementations in terms of both speed and data traffic – being faster than the **SAEV**-based svs method; and requiring less data to be transferred per query than the pure **CSRD**-mode byte-coded index. It also offers a clear tradeoff between query speed and data transfer cost. With $k = 32$ the stored index is a little larger and per-query data costs are a little higher than with $k = 8$, but query throughput approximately doubles.

6. CONCLUSIONS

Three distinct representations for sets and for computing set intersections have been examined, and experiments carried out to quantify the usefulness of each. The particular application considered is that of finding the conjunction of a set of query terms, where the sets being intersected represent the identifiers of web pages in a crawl of the `.gov` domain, and the conjunction of the terms identifies the web pages that contain all of the query terms. The query streams used are both drawn from web search engine logs.

The key findings from this work are that:

—Storing the sets as sorted arrays of explicit values (**SAEV** format) allows fast intersections to be computed, with the cost determined more by the frequency of the rarest term in the query than by another other factors, including the number of terms in the query. Of the various algorithms for working with **SAEV** representations, the traditional svs method provides fast execution, but the new max method described in this paper is almost as fast, and requires fewer **F-SEARCH** calls.

—But, as is already well known, the total storage requirement for **SAEV**-format index information is high.

—Storing the sets as compact sequences of relative differences (**CSRD** format) is much less costly in terms of storage space, but makes **MEMBER** operations more expensive, and hence intersection operations slower. Even when an auxiliary index of the type described in this work is added, querying time remains slower than with the **SAEV** format and svs approach.

—Storing the sets as bitvectors (pure **BITV** format) is expensive in terms of total index storage space, but for typical query sequences allows fast computation of intersection, with data transfer requirements that are not dissimilar to those associated with **CSRD** format representations.

—A hybrid index arrangement combining **BITV** representation for a minority of very common terms, and **CSRD** representation for the majority of less common terms, provides economical storage of the total index, and also allows very fast execution of queries, even long ones. In addition, the average per-query volume of index data that must be

transferred from disk is around half of the best of any of the other approaches that were tested.

These finding have immediate implications in document search and retrieval applications, which often select answers based on conjunction, with the eventual ranking determined by the static factors such as assessed page quality, page rank, and so on, that are used to order the documents in the collection.

An area that we have not explored in these experiments is that of phrase querying. When some or all of the query terms must appear consecutively for a document to match, more elaborate positional indexing is required, so that word locations within documents can also be compared before answer documents are identified. Even so, the first step in processing such queries is to determine whether the set of query terms co-exist in the document in a conjunctive sense. Once that has been determined, a secondary index of word positions for the terms in that document can be used to establish whether or not the phrase itself also occurs. That is, the techniques described in this paper are also applicable to phrase and other complex querying tasks.

REFERENCES

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

BARBAY, J. AND KENYON, C. 2002. Adaptive intersection and *t*-threshold problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, D. Eppstein, Ed. 390–399.

BARBAY, J., LÓPEZ-ORTIZ, A., AND LU, T. 2006. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, C. Àlvarez and M. J. Serna, Eds. LNCS, vol. 4007. 146–157.

BENTLEY, J. AND YAO, A. C.-C. 1976. An almost optimal algorithm for unbounded searching. *Information Processing Letters 5*, 3 (August), 82–87.

BLANDFORD, D. K. AND BLELLOCH, G. E. 2004. Compact representations of ordered sets. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, J. I. Munro, Ed. 11–19.

BRODNIK, A. AND MUNRO, J. 1999. Membership in constant time and almost-minimum space. *SIAM Journal on Computing 28*, 5, 1627–1640.

CLARK, D. 1996. Compact PAT trees. Ph.D. thesis, University of Waterloo.

CLAUDE, F. AND NAVARRO, G. 2008. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, A. Amir, A. Turpin, and A. Moffat, Eds. LNCS, vol. 5280. Springer, 176–187.

DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2000. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*. 743–752.

ELIAS, P. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory IT-21*, 2 (March), 194–203.

GALLAGER, R. G. AND VAN VOORHIS, D. C. 1975. Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory IT-21*, 2 (March), 228–230.

GOLOMB, S. W. 1966. Run-length encodings. *IEEE Transactions on Information Theory IT-12*, 3 (July), 399–401.

GONNET, G. H., ROGERS, L. D., AND GEORGE, J. A. 1980. An algorithmic and complexity analysis of iterpolation search. *Acta Informatica 13*, 1 (January), 39–52.

GUPTA, A., HON, W.-K., SHAH, R., AND VITTER, J. S. 2006. Compressed dictionaries: Space measures, data sets, and experiments. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, C. Àlvarez and M. J. Serna, Eds. LNCS, vol. 4007. 158–169.

HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer architecture: A quantitative approach*, fourth ed. Morgan Kaufmann, San Francisco, CA.

HWANG, F. K. AND LIN, S. 1972. A simple algorithm for merging two disjoint linearly ordered list. *SIAM Journal on Computing 1*, 31–39.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1989)*. 549–554.

MOFFAT, A. AND PORT, G. S. 1990. Splay tree melding: Theme and variations. In *Proceedings of the 13th Australasian Computer Science Conference*. Monash University, 275–284.

MOFFAT, A. AND STUIVER, L. 2000. Binary interpolative coding for effective index compression. *Information Retrieval 3,* 1 (July), 25–47.

MOFFAT, A. AND TURPIN, A. 2002. *Compression and Coding Algorithms*, first ed. Kluwer Academic Publisher, Boston.

MOFFAT, A. AND ZOBEL, J. 1996. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14,* 4, 349–379.

MUNRO, J. I. 1996. Tables. In *Proceedings of the 16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, V. Chandru and V. Vinay, Eds. LNCS, vol. 1180. Springer, 37–42.

OKANOHARA, D. AND SADAKANE, K. 2007. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*. 60–70.

PAGH, R. 2001. Low redundancy in static dictionaries with constant time query. *SIAM Journal on Computing 31,* 2, 353–363.

RAMAN, R., RAMAN, V., AND RAO, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, D. Eppstein, Ed. Society for Industrial and Applied Mathematics, 233–242.

SADAKANE, K. AND GROSSI, R. 2006. Squeezing succinct data structures into entropy bounds. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*. ACM Press, 1230–1239.

SANDERS, P. AND TRANSIER, F. 2007. Intersection in integer inverted indices. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*. 71–83.

STROHMAN, T. AND CROFT, W. B. 2007. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2007)*. ACM Press, 175–182.

TRANSIER, F. AND SANDERS, P. 2008. Compressed inverted indexes for in-memory search engines. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX 2008)*, J. I. Munro and D. Wagner, Eds. 3–12.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, second ed. Morgan Kaufmann, San Francisco.

ZOBEL, J. AND MOFFAT, A. 2006. Inverted files for text search engines. *ACM Computing Surveys 38,* 2, 6–1 – 6–56.

Received October 2009; revised April 2010; accepted July 2010.