Experiments on Adaptive Set Intersections for Text Retrieval Systems

Erik D. Demaine[1], Alejandro López-Ortiz[2], and J. Ian Munro[1]

[1] Department of Computer Science, University of Waterloo,
Waterloo, Ontario N2L 3G1, Canada,
{eddemaine,imunro}@uwaterloo.ca

[2] Faculty of Computer Science, University of New Brunswick,
P. O. Box 4400, Fredericton, N. B. E3B 5A3, Canada,
alopez-o@unb.ca

Abstract. In [3] we introduced an adaptive algorithm for computing the intersection of k sorted sets within a factor of at most $8k$ comparisons of the information-theoretic lower bound under a model that deals with an encoding of the shortest proof of the answer. This adaptive algorithm performs better for "burstier" inputs than a straightforward worst-case optimal method. Indeed, we have shown that, subject to a reasonable measure of instance difficulty, the algorithm adapts optimally up to a constant factor. This paper explores how this algorithm behaves under actual data distributions, compared with standard algorithms. We present experiments for searching 114 megabytes of text from the World Wide Web using 5,000 actual user queries from a commercial search engine. From the experiments, it is observed that the theoretically optimal adaptive algorithm is not always the optimal in practice, given the distribution of WWW text data. We then proceed to study several improvement techniques for the standard algorithms. These techniques combine improvements suggested by the observed distribution of the data as well as the theoretical results from [3]. We perform controlled experiments on these techniques to determine which ones result in improved performance, resulting in an algorithm that outperforms existing algorithms in most cases.

1 Introduction

In SODA 2000 [3] we proposed an adaptive algorithm for computing the intersection of sorted sets. This problem arises in many contexts, including data warehousing and text-retrieval databases. Here we focus on the latter application, specifically web search engines. In this case, for each keyword in the query, we are given the set of references to documents in which it occurs, obtained quickly by an appropriate data structure [1,5,9]. Our goal is to identify those documents containing all the query keywords. Typically these keyword sets are stored in some natural order, such as document date, crawl date, or by URL identifier. In practice, the sets are large. For example, as of July 2000, the average word from user query logs matches approximately nine million documents on the Google web search engine. Of course, one would hope that the answer to the query is small, particularly if the query is an intersection. It may also be expected that in dealing with grouped documents such as news articles or web sites, one will find a large number of references to one term over a few relatively short intervals of documents, and little outside these intervals. We refer to this data nonuniformity as "burstiness."

An extreme example that makes this notion more precise arises in computing the intersection of two sorted sets of size n. In this case, it is necessary to verify the total order of the elements via comparisons. More precisely, for an algorithm to be convinced it has the right answer, it should be able to *prove* that it has the right answer by demonstrating the results of certain comparisons. At one extreme, if the sets interleave perfectly, $\Omega(n)$ comparisons are required to prove that the intersection is empty. At the other extreme, if all the elements in one set precede all the elements in the other set, a single comparison suffices to prove that the intersection is empty. In between these extremes, the number of comparisons required is the number of "groups" of contiguous elements from a common set in the total order of elements. The fewer the groups the *burstier* the data.

This example leads to the idea of an *adaptive* algorithm [2,4,10]. Such an algorithm makes no a priori assumptions about the input, but determines the kind of instance it faces as the computation proceeds. The running time should be reasonable for the particular instance—not the overall worst-case.

In the case of two sets, it is possible to obtain a running time that is roughly a logarithmic factor more than the minimum number of comparisons needed for a proof *for that instance*. This logarithmic factor is necessary on average. Intersection of several sorted sets becomes more interesting because then it is no longer necessary to verify the total order of the elements. Nonetheless, in [3], we demonstrate a simple algorithmic characterization of the proof with the fewest comparisons. Another difference with $k > 2$ sets is that there is no longer an adaptive algorithm that matches the minimum proof length within a roughly logarithmic factor; it can be necessary to spend roughly an additional factor of k in comparisons [3]. Although we have been imprecise here, the exact lower bound can be matched by a fairly simple adaptive algorithm described in [3]. The method proposed, while phrased in terms of a pure comparison model, is immediately applicable to any balanced tree (e.g., B-tree) model.

This means that while in theory the advantages of the adaptive algorithm are undeniable—it is no worse than the worst-case optimal [6,7] and it does as well as theoretically possible—in practice the improvement depends on the burstiness of the actual data. The purpose of this paper is to evaluate this improvement, which leads to the following questions: what is a reasonable model of data, and how bursty is that data?

Our results are experiments on "realistic" data, a 114-megabyte crawl from the web and 5,000 actual user queries made on the Google$^{\text{TM}}$ search engine; see Section 2 for details.

What do we measure of this data? We begin by comparing two algorithms for set intersection: the optimal adaptive algorithm from [3], and a standard algorithm used in some search engines that has a limited amount of adaptiveness already, making it a tough competitor. We refer to the former algorithm as Adaptive, and to the latter algorithm as SvS, small versus small, because it repeatedly intersects the two smallest sets. As a measure of burstiness, we also compute the fewest comparisons required just to prove that the answer is correct. This value can be viewed as the number of comparisons made by an omniscient algorithm that knows precisely where to make comparisons, and hence we call it Ideal. It is important to keep in mind that this lower bound is not even achievable: there are two factors unaccounted, one required and roughly logarithmic, and the other roughly k in the worst case, not to mention any constant factors implicit in the algorithms. (Indeed we have proved stronger lower bounds in [3].) We also implement a metric called IdealLog that approximately incorporates the necessary logarithmic factor. See Section 3 for descriptions of these algorithms.

In all cases, we measure the number of comparisons used by the algorithms. Of course, this cost metric does not always accurately predict running time, which is of the most practical interest, because of caching effects and data-structuring overhead. However, the data structuring is fairly simple in both algorithms, and the memory access patterns have similar regularities, so we believe that our results are indicative of running time as well. There are many positive consequences of comparison counts in terms of reproducability, specifically machine-independence and independence from much algorithm tuning. Comparison counts are also inherently interesting because they can be directly compared with the theoretical results in [3].

Our results regarding these algorithms (see Section 4) are somewhat surprising in that the standard algorithm outperforms the optimal adaptive algorithm in many instances, albeit the minority of instances. This phenomenon seems to be caused by the overhead of the adaptive algorithm repeatedly cycling through the sets to exploit any obtainable shortcuts. Such constant awareness of all sets is necessary to guarantee how well the algorithm adapts. Unfortunately, it seems that for this data set the overhead is too great to improve performance on average, for queries with several sets. Thus in Section 5 we explore various compromises between the two algorithms, to evaluate which adaptive techniques have a globally positive effect. We end up with a partially adaptive algorithm that outperforms both the adaptive and standard algorithms in most cases.

2 The Data

Because our exploration of the set-intersection problem was motivated by text retrieval systems in general and web search engines in particular, we tested the algorithm on a 114-megabyte subset of the World Wide Web using a query log from Google. The subset consists of 11,788,110 words[1], with 515,277 different

[1] The text is tokenized into "words" consisting of alphanumerical characters; all other characters are considered whitespace.

Table 1. Query distribution in Google log.

# keywords	# queries in log
2	1,481
3	1,013
4	341
5	103
6	57
7	26
8	14
9	4
10	2
11	1

words, for an average of 22.8 occurrences per word. Note that this average is in sharp contrast to the average number of documents containing a query word, because a small number of very common words are used very often.

We indexed this corpus using an inverted word index, which lists the document(s) in which each term occurs. The plain-text word index is 48 megabytes.

The query log is a list of 5,000 queries as recorded by the Google search engine. Queries consisting of just a single keyword were eliminated because they require no intersections. This reduces the query set to 3,561 entries. Of those, 703 queries resulted in trivially empty sets because one or more of the query terms did not occur in the index at all. The remaining 2,858 queries were used to test the intersection algorithms. Table 1 shows the distribution of the number of keywords per query. Note that the average number of keyword terms per query is 2.286, which is in line with data reported elsewhere for queries to web search engines [8]. Notice that beyond around seven query terms the query set is not large enough to be representative.

The data we use is realistic in the sense that it comes from a real crawl of the web and a real query log. It is idisosincratic in the sense that it is a collection of pages, grouped by topic, time and language. Other set intersections outside text retrieval, or even other text-retrieval intersections outside the web, might not share these characteristics.

The query log has some anomalies. First, the Google search engine does not search for stop words in a query unless they are preceded by '+'. This may cause knowledgeable users to refrain from using stop words, and thus produce an underrepresentation of the true frequency of stop words in a free-form search engine. Second, it seems that in the Google logs, all stop words have been replaced by a canonical stop word 'a'. Third, the lexically last query begins with 'sup', so for example no queries start with 'the' (which is not a Google stop word).

Figure 1 shows how the different set sizes are represented in the query log. At the top of the chart we see a large set corresponding to the word 'a', which is very common both in the corpus and in the query log. In Figure 2 we see the distribution of the total set sizes of the queries. In other words, given a

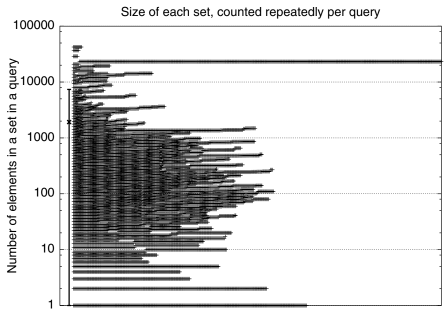


Fig. 1. Size of each set, counted repeatedly per query.

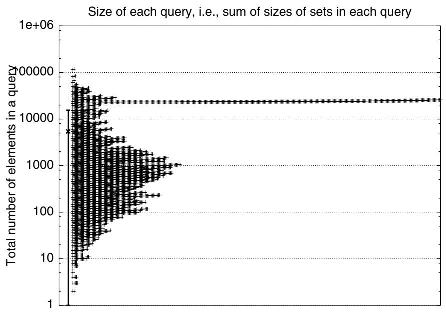


Fig. 2. Total number of elements involved in each query.

query from the query log, we sum the number of elements in each of the sets in that query and plot this value. As it is to be expected from the sum of a set of random variables, the distribution roughly resembles a normal distribution, with the exception of those queries involving the stop word ‘a’, as discussed above.

3 Main Algorithms

We begin by studying three main methods in the comparison model for determining the intersection of sorted sets. The first algorithm, which we refer to as SvS, repeatedly intersects the two smallest sets; to intersect a pair of sets, it binary searches in the larger set for each element in the smaller set. A more formal version of the algorithm is presented below.

Algorithm SvS.

```
– Sort the sets by size.
– Initially let the candidate answer set be the smallest set.
– For every other set $S$ in increasing order by size:
    • Initially set $\ell$ to 1.
    • For each each element $e$ in the candidate answer set:
        ◦ Perform a binary search for $e$ in $S$, between $\ell$ and $|S|$.[2]
        ◦ If $e$ is not found, remove it from the candidate answer set.
        ◦ Set $\ell$ to the value of *low* at the end of the binary search.
```

This algorithm is widely used in practice, and even possesses a certain amount of adaptivity. Because intersections only make sets smaller, as the algorithm progresses with several sets, the time to do each intersection effectively reduces. In particular, the algorithm benefits largely if the set sizes vary widely, and

[2] John Bentley (personal communication, September 2000) has pointed out that it is frequently more efficient to binary search between 1 and $|S|$ all the time, because of similar access patterns causing good cache behavior. However, since we are working in the comparison model, searching from ℓ can only make SvS a stonger competitor.

performs poorly if the set sizes are all roughly the same. More precisely, the algorithm SvS makes at least $\Omega(r \log n)$ and at most $O(n \log(n/k))$ comparisons, where r is the size of the resulting intersection and n is the total number of elements over all k sets.

The second algorithm, which we refer to as Adaptive, is the adaptive method proposed by the authors [3]. It has two main adaptive features. The most prominent is that the algorithm takes an element (the smallest element in a particular set whose status in the intersection is unknown) and searches for it in each of the other sets "simultaneously," and may update this candidate value in "mid search." A second adaptive feature is the manner in which the algorithm performs this search. It uses the well-known approach of starting at the beginning of an array and doubling the index of the queried location until we overshoot. A binary search between the last two locations inspected completes the search for a total time of $2 \lg i$ comparisons, where i denotes the final location inspected. We refer to this approach as *galloping*.

A more precise description of the algorithm is the following:

Algorithm Adaptive.

- Initially set the *eliminator* to the first element of the first set.
- Repeatedly cycle through the sets:
  - Perform one galloping step in the current set.
  - If we overshoot:
    - Binary search to identify the precise location of the eliminator.
    - If present, increase occurrence counter and output if the count reaches k.
    - Otherwise, set the new eliminator to the first element in the current set that is larger than the current eliminator. If no such element exists, exit loop.

In [3], Adaptive is described as working from both ends of each set, but for simplicity we do not employ this feature at all in this work. The worst-case performance of Adaptive is within a factor of at most $O(k)$ of any intersection algorithm, on average, and its best-case performance is within a roughly logarithmic factor of the "offline ideal method."

This last metric, which we refer to as Ideal, measures the minimum number of comparisons required in a *proof* of the intersection computed. Recall that an intersection proof is a sequence of comparisons that uniquely determines the result of the intersection. For example, given the sorted sets $\{1, 3\}$ and $\{2\}$, the comparisons $(1 < 2)$ and $(2 < 3)$ form a proof of the emptiness of the intersection. Of course, computing the absolute smallest number of comparisons required takes significantly more comparisons than the value itself, but it can be computed in linear time as proved in [3].

Because any algorithm produces a proof, the smallest descriptive complexity of a proof for a given instance is a lower bound on the time complexity of the intersection of that instance. Unfortunately this descriptive complexity or Kolmogorov complexity is not computable, so we cannot directly use this lower bound as a measure of instance difficulty or burstiness. Instead we employ two approximations to this lower bound.

First observe that the number of comparisons alone (as opposed to a binary encoding of which comparisons) is a lower bound on the descriptive complexity of a proof. This is precisely the Ideal metric. The adaptive algorithm takes a roughly logarithmic factor more than Ideal, and it may take roughly a factor of k longer in the worst case. However, Ideal provides a baseline unachievable optimum, similar to that used in online competitive analysis.

This baseline can be refined by computing the complexity of a description of this proof. Specifically, we describe a proof by encoding the compared elements, for each element writing the set and displacement from the previously compared element in that set. To encode this gap we need, on average, the log of the displacement value, so we term this the log-gap metric. For the purposes of this work we ignore the cost of encoding which sets are involved in each comparison.

In SODA we show that a log-gap encoding is efficient, using information-theoretic arguments. As we mentioned above, Ideal can be found in linear time, yet the shortest proof even by the log-gap metric seems difficult to compute. One can estimate this value, though, by computing the log-gap encoding of the proof with the fewest comparisons. This leads to a metric called IdealLog, Ideal with a log gap. We cannot claim that this is the shortest proof description but it seems a reasonable approximation.

4 Main Experimental Results

Figure 3 shows the number of comparisons required by Ideal to show that the intersection is empty, and Figure 4 shows the size of the log-gap encoding of this proof. The integral points on the x-axis correspond to the number of terms per query, and the y-axis is the number of comparisons taken by either metric on a logarithmic scale. Within each integral gap on the x-axis is a frequency histogram. Each cross represents a query. The crosses are spaced horizontally to be vertically separated by at least a constant amount, and they are scaled to fill the horizontal space available. Thus, at a given vertical level, the width

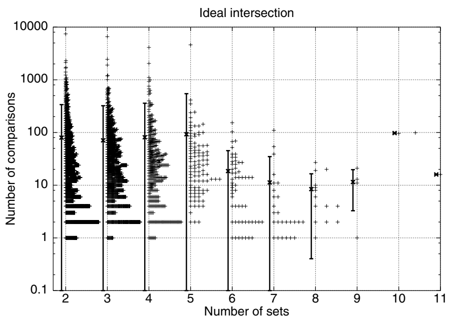


Fig. 3. Number of comparisons of Ideal by terms in query.

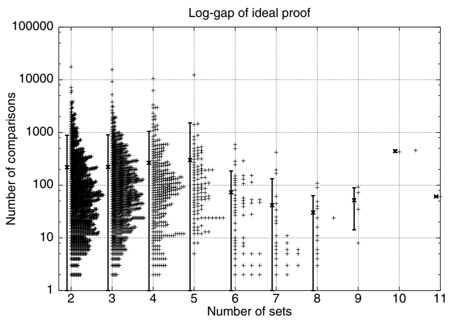


Fig. 4. Encoding size of IdealLog by terms in query.

of the chart approximates the relative frequency, and the density of the chart is indicative of the number of queries with that cost.[3] In addition, to the left of each histogram is a bar (interval) centered at the mean (marked with an 'X') and extending up and down by the standard deviation. This histogram/bar format is used in most of our charts.

Figure 5 shows the number of comparisons used by Adaptive to compute the intersection, with axes as in Figure 3. These values are normalized in Figure 7 by dividing by the IdealLog metric for each query. We observe Adaptive requires on the average about $1 + 0.4k$ times as many comparisons as IdealLog. This observation matches the worst-case ratio of around $\Theta(k)$, suggesting that Adaptive is wasting time cycling through the sets.

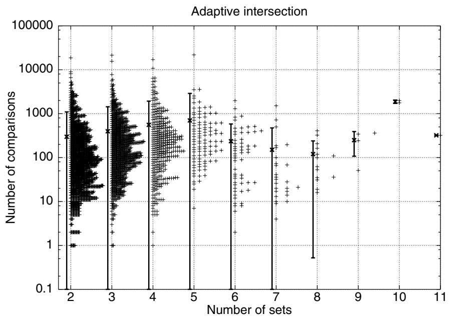


Fig. 5. Number of comparisons of Adaptive by terms in query.

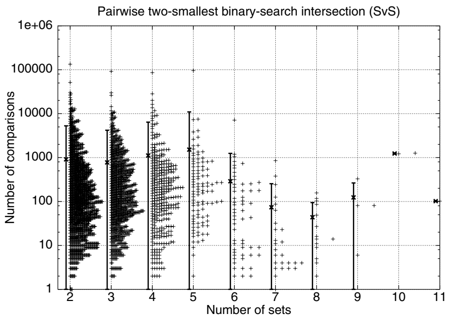


Fig. 6. Numbers of comparisons of SvS by terms in query.

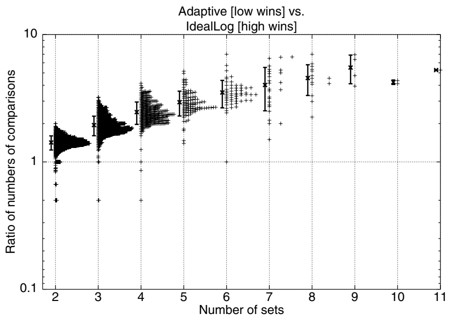


Fig. 7. Ratio of number of comparisons of Adaptive over IdealLog.

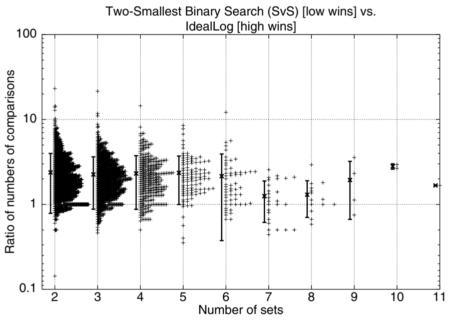


Fig. 8. Ratio of number of comparisons of SvS over IdealLog.

[3] Unfortunately, there is a limit to the visual density, so that, for example, the leftmost histograms in Figures 3 and 4 both appear black, even though the histogram in Figure 3 is packed more tightly because of many points with value 2 (coming from queries with 1-element sets—see Figure 1).

Figures 6 and 8 show the same charts for SvS, as absolute numbers of comparisons and as ratios to IdealLog. They show that SvS also requires a substantially larger amount of comparisons than IdealLog, and for few sets (2 or 3) often more comparisons than Adaptive, but the dependence on k is effectively removed. In fact, SvS appears to improve slightly as the number of sets increases, presumably because more sets allows SvS's form of adaptivity (removing candidate elements using small sets) to become more prominent.

Figure 9 shows the ratio of the running times of Adaptive and SvS, computed individually for each query. Figure 10 shows the difference in another way, subtracting the two running times and normalizing by dividing by IdealLog. Either way, we see directly that Adaptive performs frequently better than SvS only for a small number of sets (2 or 3), presumably because of Adaptive's overhead in cycling through the sets. SvS gains a significant advantage because the intersection of the smallest two sets is very small and indeed often empty, and therefore SvS often terminates after the first pass, having only examined two sets, while Adaptive constantly examines all k sets.

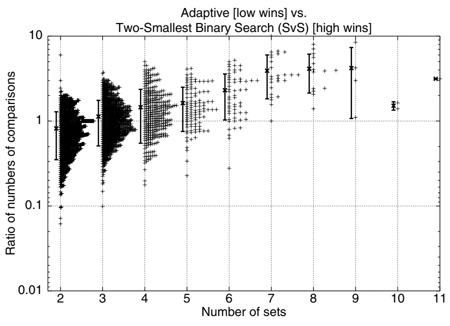


Fig. 9. Ratio of number of comparisons of Adaptive over SvS.

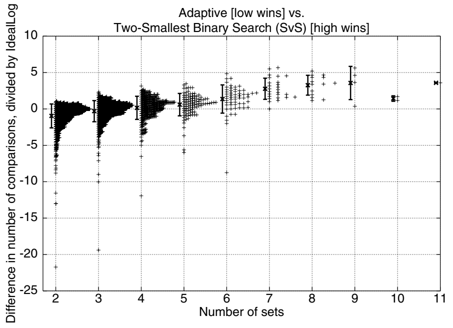


Fig. 10. Difference in number of comparisons of Adaptive and SvS, normalized by IdealLog.

5 Further Experiments

In this section we explore various compromises between the adaptive algorithm and SvS to develop a new algorithm better than both for any number of sets. More precisely, we decompose the differences between the two algorithms into main techniques. To measure the relative effectiveness of each of these techniques we examine most (though not all) of the possible combinations of the techniques.

The first issue is how to search for an element in a set. Binary search is optimal when trying to locate a random element. However, in the case of computing an intersection using SvS (say), on the average the element being located is likelier to be near the front of the array. Therefore starting the search from the front, as galloping does, is a natural improvement. Figure 11 confirms that galloping in the second-smallest set ("half galloping") is usually better than binary searching (SvS). Variations in galloping may also result in improvements; one simple example is increasing the galloping factor from 2 to 4. This particular change has no substantial effect, positive or negative, in the case of half galloping; see Figure 12. Another natural candidate is the Hwang-Lin merging algorithm for optimal intersection of two random sets [6,7]. Again, comparing to the half-galloping method, there is no clear advantage either way; see Figure 13.

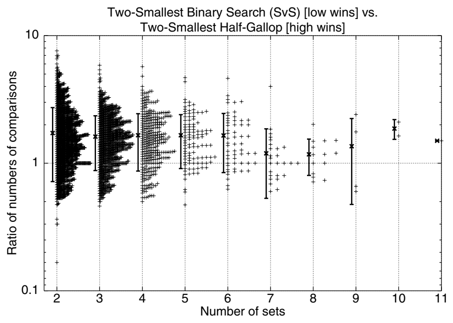


Fig. 11. Ratio of number of comparisons of SvS over SvS with galloping (Two-Smallest Half-Gallop).

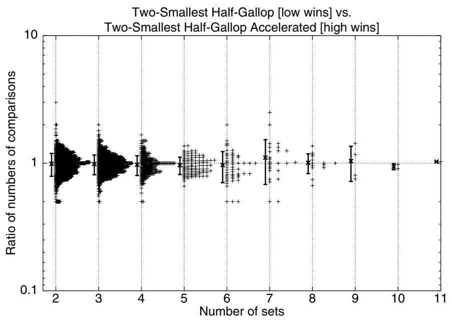


Fig. 12. Ratio of number of comparisons of Two-Smallest Half-Gallop (factor 2) over Two-Smallest Half-Gallop Accelerated (factor 4).

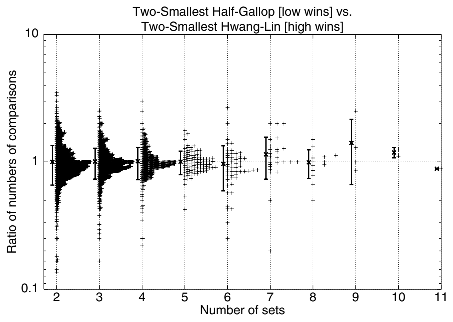


Fig. 13. Ratio of number of comparisons of Two-Smallest Half-Gallop over Two-Smallest Hwang-Lin.

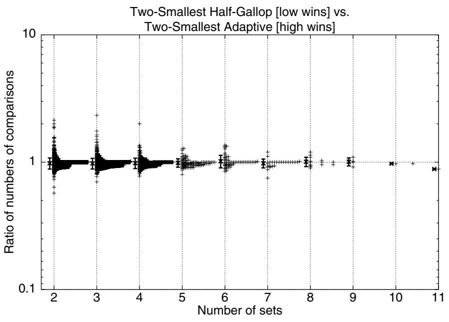


Fig. 14. Ratio of number of comparisons of Two-Smallest Half-Gallop over Two-Smallest Adaptive.

A second issue is that SvS sequentially scans the smallest set, and for each element searches for a matching element in the second-smallest set. Alternatively, one could alternate the set from which the candidate element comes from. In one step we search in the second-smallest set for the first element in the smallest set; in the next step we search in the smallest set for the first uneliminated element in the second-smallest set; etc. This is equivalent to repeatedly applying the adaptive algorithm to the smallest pair of sets, and hence we call the algorithm Two-Smallest Adaptive. The results in Figure 14 show that this galloping in both sets rarely performs differently from galloping in just one set, but when there is a difference it is usually an improvement.

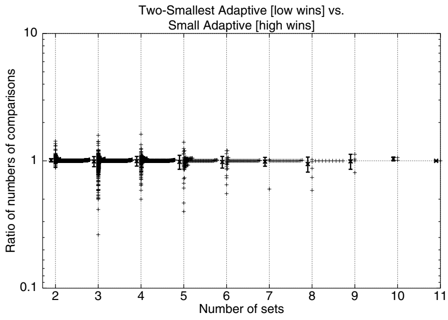
A third issue is that Adaptive performs galloping steps cyclically on all sets. This global awareness is necessary to guarantee that the number of comparisons is within a factor of optimal, but can be inefficient, especially considering that in our data the pairwise intersection of the smallest two sets is often empty. On the other hand, SvS blindly computes the intersection of these two sets with no lookahead. One way to blend the approaches of Adaptive and SvS is what we call Small Adaptive. We apply a galloping binary search (or the first step of Hwang-Lin) to see how the first element of the smallest set fits into the second-smallest set. If the element is in the second-smallest set, we next see whether it is in the third-smallest and so on until we determine whether it is in the answer. (Admittedly this forces us to make estimates of the set size if we use Hwang-Lin.)

This method does not increase the work from SvS because we are just moving some of the comparisons ahead in the schedule of SvS. The advantage, though, is that this action will eliminate arbitrary numbers of elements from sets and so will change their relative sizes. Most notably it may change which are the smallest two sets, which would appear to be a clear advantage.

Thus, if we proceed in set-size order, examining the remaining sets, this approach has the advantage that the work performed is no larger than SvS, and on occasion it might result in savings if another set becomes completely eliminated. For example, if we are intersecting the sets $A_1 = \{3, 6, 8\}$, $A_2 = \{4, 6, 8, 10\}$ and $A_3 = \{1, 2, 3, 4, 5\}$, we start by examining A_1 and A_2, we discard 3 and 4, and identify 6 as a common element. SvS would carry on in these two sets obtaining the provisional result set $R = \{6, 8\}$ which would then be intersected against A_3. On the other hand an algorithm that immediately examines the remaining sets would discover that all elements in A_3 are smaller than 6 and immediately report that the entire intersection is empty.

Another source of improvement from examining the remaining sets once a common element has been identified is that another set might become the smallest. For example, let A_1 and A_2 be as in the previous example and let $A'_3 = \{1, 2, 3, 4, 5, 9\}$. Then after reaching the common element 6 in A_1 and A_2 the algorithm examines A_3 and eliminates all but $\{9\}$. At this stage the remaining elements to be explored are $\{8\}$ from A_1, $\{8, 10\}$ from A_2 and $\{9\}$ from A_3. Now the new two smallest sets are A_1 and A_3 and it proceeds on these two sets.

As a final tuning, if the two smallest sets do not change, Small Adaptive searches alternately between the two sets, as in Two-Smallest Adaptive. Thus the only difference between these two algorithms is when the two smallest sets intersect. On the web dataset, this happens so infrequently that the difference between Small Adaptive and Two-Smallest Adaptive is slight; see Figure 15. However, when Small Adaptive makes a difference it is usually an improvement, and there are several instances with a fairly large improvement.



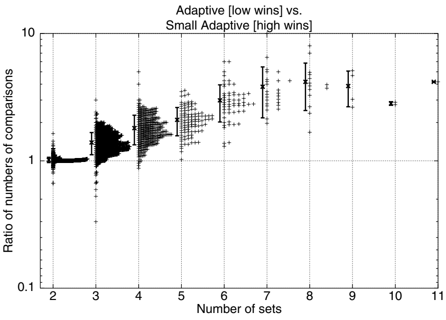


Fig. 15. Ratio of number of comparisons of Two-Smallest Adaptive over Small Adaptive.

Fig. 16. Ratio of number of comparisons of Adaptive over Small Adaptive.

We have reached the conclusion that four techniques have positive impact: galloping, alternating between the two smallest sets, advancing early to additional sets when a common element is encountered (a limited form of adaptivity), and updating which sets are smallest. The techniques which had little effect, positive or negative, were the Hwang-Lin replacement for galloping, and accelerating galloping. The only technique with significant negative impact is the full-blown adaptivity based on cycling through all the sets.

We designed Small Adaptive by starting with SvS and incorporating many essential features from Adaptive to improve past SvS. In particular, Figures 11, 14, and 15 have shown that Small Adaptive wins over SvS. (This can also be verified directly.) But how does Small Adaptive compare to our other "extreme," the adaptive algorithm from [3]? Surprisingly, Figure 16 shows that Small Adaptive almost always performs better than Adaptive, regardless of the number of sets (unlike SvS which was incomparable with Adaptive).

Table 2 summarizes the algorithms encountered, and a few other possible combinations. Table 3 shows the average overall running times for these algorithms, as well as the standard deviations. Interestingly, in this aggregate metric, Adaptive outperforms SvS; this is because many queries have only 2 or 3 sets. In addition, the algorithm with the smallest average running time is Small Adaptive. We conclude that Small Adaptive seems like the best general algorithm for computing set intersections based on these ideas, for this dataset.

6 Conclusion

In this paper we have measured the performance of an optimally adaptive algorithm for computing set intersection against the standard SvS algorithm and an offline optimal Ideal. From this measurement we observed a class of instances in which Adaptive outperforms SvS. The experiments then suggest several avenues for improvement, which were tested in an almost orthogonal fashion. From these additional results we determined which techniques improve the performance of

Table 2. Algorithm characteristics key table.

Algorithm	Cyclic/2 Smallest	Symmetric	Update Smallest	Advance on Common Elt.	Gallop Factor
Adaptive	Cyclic	Y	—	—	2
Adaptive 2	Cyclic	Y	—	—	4
Ideal	—	—	—	—	—
Small Adaptive	Two	Y	Y	Y	2
Small Adaptive Accel.	Two	Y	Y	Y	4
Two-Smallest Adaptive	Two	Y	N	N	2
Two-Smallest Adaptive Accel.	Two	Y	N	N	4
Two-Smallest Binary Search (SvS)	Two	N	N	N	—
Two-Smallest Half-Gallop	Two	N	N	N	2
Two-Smallest Half-Gallop Accel.	Two	N	N	N	4
Two-Smallest Hwang-Lin	Two	N	N	N	—
Two-S'est Smart Binary Search	Two	N	Y	N	—
Two-S'est Smart Half-Gallop	Two	N	Y	N	2
Two-S'est Smart Half-Gallop Accel.	Two	N	Y	N	4

Table 3. Aggregate performance of algorithms on web data.

Algorithm	Average	Std. Dev.	Min	Max
Adaptive	371.46	1029.41	1	21792
Adaptive Accelerated	386.75	1143.65	1	25528
Ideal	75.44	263.37	1	7439
Small Adaptive	315.10	962.78	1	21246
Small Adaptive Accelerated	326.58	1057.41	1	24138
Two-Smallest Adaptive	321.88	998.55	1	22323
Two-Smallest Adaptive Accelerated	343.90	1153.32	1	26487
Two-Smallest Binary Search (SvS)	886.67	4404.36	1	134200
Two-Smallest Half-Gallop	317.60	989.98	1	21987
Two-Smallest Half-Gallop Accelerated	353.66	1171.77	1	27416
Two-Smallest Hwang-Lin	365.76	1181.58	1	25880
Two-Smallest Smart Binary Search	891.36	4521.62	1	137876
Two-Smallest Smart Half-Gallop	316.45	988.25	1	21968
Two-Smallest Smart Half-Gallop Accelerated	350.59	1171.43	1	27220

an intersection algorithm for web data. We blended theoretical improvements with experimental observations to improve and tune intersection algorithms for the proposed domain. In the end we obtained an algorithm that outperforms the two existing algorithms in most cases. We conclude that these techniques are of practical significance in the domain of web search engines.

Acknowledgments

We thank Monika Henzinger of Google for access to the query logs. This work was supported by NSERC.

References

1. R. Baeza-Yates. *Efficient Text Searching.* PhD thesis, Department of Computer Science, University of Waterloo, 1989.
2. Svante Carlsson, Christos Levcopoulos, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9:629–648, 1993.
3. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, San Francisco, California, January 2000.
4. Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1992.
5. William Frakes and Richardo Baeza-Yates. *Information Retrieval.* Prentice Hall, 1992.
6. F. K. Hwang. Optimal merging of 3 elements with n elements. *SIAM Journal on Computing*, 9(2):298–320, 1980.
7. F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1980.
8. Michael Lesk. "Real world" searching panel at SIGIR 1997. *SIGIR Forum*, 32(1), Spring 1998.
9. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searchs. In *Proceedings of the 1st Symposium on Discrete Algorithms*, pages 319–327, 1990.
10. Alistair Moffat, Ola Petersson, and Nicholas C. Wormald. A tree-based Mergesort. *Acta Informatica*, 35(9):775–793, August 1998.