Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences

Ricardo Baeza-Yates and Alejandro Salinger

Center for Web Research
Department of Computer Science
University of Chile
Blanco Encalada 2120
Santiago, Chile.

Abstract. This work presents an experimental comparison of intersection algorithms for sorted sequences, including the recent algorithm of Baeza-Yates. This algorithm performs on average less comparisons than the total number of elements of both inputs (n and m respectively) when $n = \alpha m$ ($\alpha > 1$). We can find applications of this algorithm on query processing in Web search engines, where large intersections, or differences, must be performed fast. In this work we concentrate in studying the behavior of the algorithm in practice, using for the experiments test data that is close to the actual conditions of its applications. We compare the efficiency of the algorithm with other intersection algorithm and we study different optimizations, showing that the algorithm is more efficient than the alternatives in most cases, especially when one of the sequences is much larger than the other.

1 Introduction

In this work we study different algorithms to compute the intersection of sorted sequences. This problem is a particular case of a generic problem called multiple searching [3] (see also [15], research problem 5, page 156), which consists of, given an n-element data multiset, D, drawn from an ordered universe, search D for each element of an m-element query multiset, Q, drawn from the same universe. The found elements form, exactly, the intersection of both multisets.

The sorted sequences intersection problem finds its motivation in Web search engines, since most of them use inverted indices, where for each different word, we have a list of positions or documents where it appears. Generally, these lists are ordered by some criterion, like position, a global precomputed ranking, frequency of occurrence in a document, etc. To compute the result of a query, in most cases we need to intersect these lists. In practice these lists can have hundreds of millions of elements, hence, it is useful to have an algorithm that is fast and efficient on average.

In the case when D and Q are sets (and not multisets) already ordered, multiple search can be solved by merging both sets. However, this is not optimal for all possible cases. In fact, if m is small (say if $m = o(n/\log n)$), it is better to do

m binary searches obtaining an $O(m \log n)$ algorithm [2], where the complexity metric is the number of comparisons between any pair of elements. Baeza-Yates' algorithm matches both complexities depending on the value of m. On average, it performs less than $m+n$ comparisons when both sets are ordered under some pessimistic assumptions.

This work focuses on the experimental study of this algorithm, as well as different optimizations to it. The experiments consisted on measuring the running time of the original algorithms and its optimizations with sequences of ordered random integer numbers and comparing it to an algorithm based on merging and to the *Adaptive* algorithm, proposed by Demaine et al. [10], which seems to be the most used in practice. Our results show that Baeza-Yates' algorithm is slightly better than Adaptive and much better than Merge when the length of the sequences differ considerably.

In Section 2 we present related work. Section 3 presents the motivation for our problem and some practical issues. Section 4 presents the algorithms used for the comparison, including the algorithm of Baeza-Yates and a proposed optimization. Section 5 presents the experimental results obtained. Throughout this paper $n \geq m$ and logarithms are base two unless explicitly stated otherwise.

2 Related Work

In order to solve the problem of determining whether any elements of a set of $n+m$ elements are equal, we require at least $\Theta((n+m)\log(n+m))$ comparisons in the worst case (see [13]). However, this lower bound does not apply to the multiple search problem nor, equivalently, to the set intersection problem. Conversely, the lower bounds of the search problem do apply to the element uniqueness problem [12]. This idea was exploited by Demaine *et al.* to define an adaptive multiple set intersection algorithm [10, 11] that finds the common elements by searching in an unbounded domain. They also define the difficulty of a problem instance, which was refined later by Barbay and Kenyon [8].

For the ordered case, lower bounds on set intersection are also lower bounds for merging both sets. However, the converse is not true, as in set intersection we do not need to find the actual position of each element in the union of both sets, just if one element is in the other set or not. Although there has been a lot of work on minimum comparison merging in the worst case, almost no research has been done on the average case because it does not make much of a difference. However, this is not true for multiple search, and hence for set intersection [3].

The algorithm of Baeza-Yates [1] adapts to the input values. In the best case, the algorithm performs $\lceil \log(m+1) \rceil \lceil \log(n+1) \rceil$ comparisons, which for $m = O(n)$, is $O(\log^2(n))$. In the worst case, the number of comparisons performed by the algorithm is

$$W(m,n) = 2(m+1)\log((n+1)/(m+1)) + 2m + O(\log(n))$$

Therefore, for small m, the algorithm is $O(m\log(n))$, while for $n = \alpha m$ it is $O(n)$. In this case, the ratio between this algorithm and merging is $2(1+$

$\log(\alpha))/(1+\alpha)$ asymptotically, being 1 when $\alpha = 1$. The worst case is worse than merging for $1 < \alpha < 6.3197$ having its maximum at $\alpha = 2.1596$, where it is 1.336 times slower than merging. Hence the worst case of the algorithm matches the complexity of both, the merging and the multiple binary search, approaches, adapting nicely to the size of m. For the average case, under pessimistic assumptions, the number of comparisons is:

$$A(m,n) = (m+1)(\ln((n+1)/(m+1)) + 3 - 1/\ln(2)) + O(\log n)$$

For $n = \alpha m$, the ratio between this algorithm and merging is $(\ln(\alpha) + 3 - 1/\ln(2))/(1+\alpha)$ which is at most 0.7913 when $\alpha = 1.2637$ and 0.7787 when $\alpha = 1$. The details of the algorithm are presented on section 4.2.

3 Motivation: Query Processing in Inverted Indices

Inverted indices are used in most text retrieval systems [4]. Logically, they are a vocabulary (set of unique words found in the text) and a list of references per word to its occurrences (typically a document identifier and a list of word positions in each document). In simple systems (Boolean model), the lists are sorted by document identifier, and there is no ranking (that is, there is no notion of relevance of a document). In that setting, an intersection algorithm applies directly to compute Boolean operations on document identifiers: union (OR) is equivalent to merging, intersection (AND) is the operation on study (we only keep the repeated elements), and subtraction implies deleting the repeated elements, which is again similar to an intersection. In practice, long lists are not stored sequentially, but in blocks. Nevertheless, these blocks are large, and the set operations can be performed in a block-by-block basis.

In complex systems ranking is used. Ranking is typically based in word statistics (number of word occurrences per document and the inverse of the number of documents having it). Both values can be precomputed and the reference lists are then stored by decreasing intra-document word frequency order to have first the most relevant documents. Lists are then processed by decreasing inverse extra-document word frequency order (that is, we process the shorter lists first), to obtain first the most relevant documents. However, in this case we cannot always have a document identifier mapping such that lists are sorted by that order. Nevertheless, they are partially ordered by identifier for all documents of equal word frequency.

The previous scheme was used initially on the Web, but as the Web grew, the ranking deteriorated because word statistics do not always represent the content and quality of a Web page and also can be "spammed" by repeating and adding (almost) invisible words. In 1998, Page and Brin [9] described a search engine (which was the starting point of Google) that used links to rate the quality of a page, a scheme called PageRank. This is called a global ranking based in popularity, and is independent of the query posed. It is out of the scope of this paper to explain PageRank, but it models a random Web surfer and the ranking of a page is the probability of the Web surfer visiting it. This probability induces a total order that can be used as document identifier. Hence, in a pure link based search engine we can use the intersection algorithm as before. However, nowadays hybrid ranking schemes that combine link and word evidence are used. In spite of this, a link based mapping still gives good results as it approximates well the true ranking (which can be corrected while is computed).

Another important type of query is sentence search. In this case we use the word position to know if a word follows or precedes a word. Hence, as usually sentences are small, after we find the Web pages that have all of them, we can process the first two words[1] to find adjacent pairs and then those with the third word and so on. This is like to compute a particular intersection where instead of finding repeated elements we try to find correlative elements (i and $i+1$), and therefore we can use again the intersection algorithm as word positions are sorted. The same is true for proximity search. In this case, we can have a range k of possible valid positions (that is $i \pm k$) or to use a different ranking weight depending on the proximity.

Finally, in the context of the Web, an adaptive algorithm is in practice much faster because the uniform distribution assumption is pessimistic. In the Web, the distribution of word occurrences is quite biased. The same is true with query frequencies. Both distributions follow a power law (a generalized Zipf distribution) [4, 6]. However, the correlation of both distributions is not high [7] and even low [5]. That implies that the average length of the lists involved in the query are not that biased. That means that the average lengths of the lists, n and m, when sampled, will satisfy $n = \Theta(m)$ (uniform), rather than $n = m + O(1)$ (power law). Nevertheless, in both cases our algorithm makes an improvement.

4 The Algorithms

Suppose that D is sorted. In this case, obviously, if Q is small, will be faster to search every element of Q in D by using binary search. Now, when Q is also sorted, set intersection can be solved by merging. In the worst or average case, straight merging requires $m + n - 1$ comparisons. However, we can do better for set intersection. Next, we describe here the different algorithms compared. We do not include the merging algorithm as it is well known.

4.1 Adaptive

This algorithm [10, 11] works as follows: we take one of the sets, and we choose its first element, which we call *elim*. We search *elim* in the other set, making exponential jumps, this is, looking at positions $1, 2, 4, \ldots, 2^i$. If we overshoot, that is, the element in the position 2^i is larger than *elim*, we binary search *elim* between positions 2^{i-1} and 2^i[2] If we find it, we add it to the result. Then, we

[1] Actually, it is more efficient to use the two words with shorter lists, and so on until we get to the largest list if the intersection is still non empty.

[2] This is the classical result of Bentley and Yao for searching an element in an unbounded set which is $O(\log n)$.

remember the position where *elim* was (or the position where it should have been) so we know that from that position backwards we already processed the set. Now we chose *elim* as the smallest element of the set that is greater than the former *elim* and we exchange roles, making jumps from the position that signals the processed part of the set. We finish when there is no element greater than the one we are searching.

4.2 Baeza-Yates

Baeza-Yates' algorithm is based on a double binary search, improving on average under some pessimistic assumptions. The algorithm introduced in [1] can be seen as a balanced version of Hwang and Lin's [14] algorithm adapted to our problem.

The algorithm works as follows. We first binary search the median (middle element) of Q in D. If found, we add that element to the result. Found or not, we have divided the problem in searching the elements smaller than the median of Q to the left of the position found on D, or the position the element should be if not found, and the elements bigger than the median to the right of that position. We then solve recursively both parts (left sides and right sides) using the same algorithm. If in any case, the size of the subset of Q to be considered is larger than the subset of D, we exchange the roles of Q and D. Note that set intersection is symmetric in this sense. If any of the subsets is empty, we do nothing.

A simple way to improve this algorithm is to apply the original algorithm not over the complete sets D and Q, but over a subset of both sets where they actually overlap, and hence, where we can really find elements that are part of the intersection.

We start by comparing the smallest element of Q with the largest of D, and the largest of Q with the smallest of D. If both sets do not overlap, the intersection is empty. Otherwise, we search the smallest and largest element of D in Q, to find the overlap, using just $O(\log m)$ time. Then we apply the previous algorithm just to the subsets that actually overlaps. This improves both, the worst and the average case. The dual case is also valid, but then finding the overlap is $O(\log n)$, which is not good for small m. This optimization is mentioned in [1], but it is effectiveness is not studied.

5 Experimental Results

We compared the efficiency of the algorithm, which we call *Intersect* in this section, with an intersection algorithm based on merging, and with an adaptation of the *Adaptive* algorithm [10, 11] for the intersection of two sequences. In addition, we show the results obtained with the optimizations of the algorithm.

We used sequences of integer random numbers, uniformly distributed in the range $[1, 10^9]$. We varied the length of one of the lists (n) from 1,000 to 22,000 with a step of 3,000. For each of these lengths we intersected those sequences with sequences of four different lengths (m), from 100 to 400. We use twenty random instances per case and ten thousand runs (to eliminate the variations due to the operating system given the small resulting times).

The programs were implemented in C using the Gcc 3.3.3 compiler in a Linux platform running an Intel(R) Xeon(TM) CPU 3.06GHz with 512 Kb cache and 2Gb RAM.

Figure 1 shows a comparison between Intersect and Merge. We can see that Intersect is better than Merge when n increases and that the time increases for larger values of m.

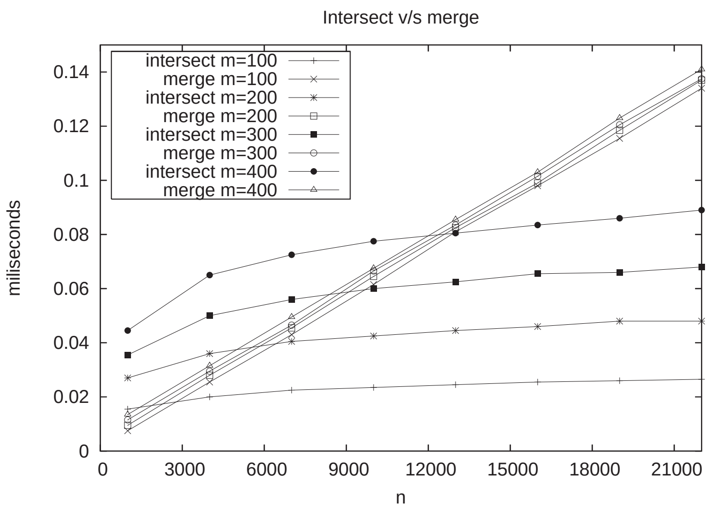

Fig. 1. Experimental results for Intersect and Merge for different values of n and m.

Figure 2 shows a comparison between the times of Intersect and Adaptive. We can see that the times of both algorithms follow the same tendency and that Intersect is better than Adaptive.

Figure 3 shows the results obtained with the Intersect algorithm and the optimization described at the end of the last section. For this comparison, we also added the computation of the overlap of both sequences to Merge.

We can see that there is no big difference between the original and the optimized algorithm, and moreover, the original algorithm was a bit faster than the optimized one. The reason why the optimization did not result in an improvement can be the uniform distribution of the test data. As the random numbers are uniformly distributed, in most cases the overlap of both sets covers a big part of Q. Then, the optimization does not produce any improvement and it only results in a time overhead due to the overlap search.

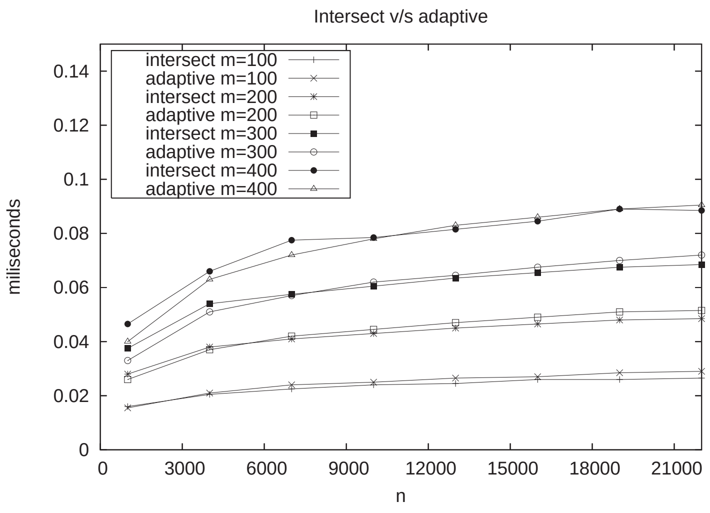

Fig. 2. Experimental results for Intersect and Adaptive, for different values of n and m.

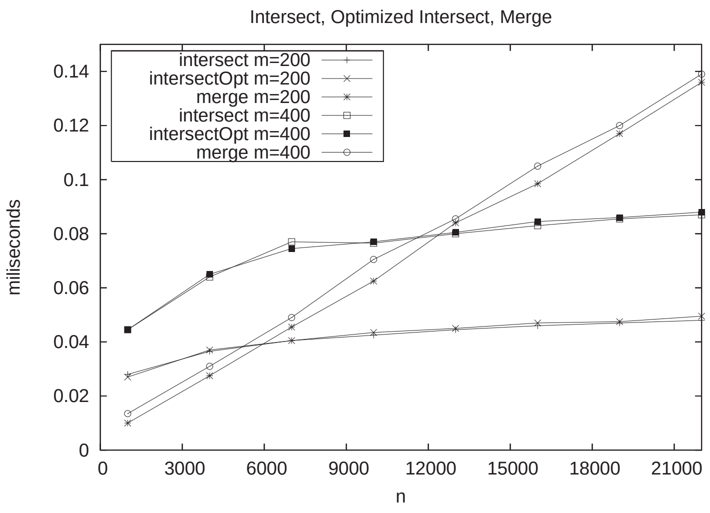

Fig. 3. Experimental results for Intersect, optimized Intersect and Merge, for different values of n and $m = 200$ y $m = 400$.

5.1 Hybrid Algorithms

We can see from the experimental results obtained that there is a section of values of n where Merge is better than Intersect. Hence, a natural idea is to combine both algorithms in one hybrid algorithm that runs each of them when convenient.

In order to know where is the cutting point to use one algorithm instead of the other, we measured for each value of n the time of both algorithms with different values of m until we identified the value of m where Merge was faster than Intersect. These values of m form a straight line as a function of n, which we can observe in Fig. 4. This straight line is approximated by $m = 0.033n + 8.884$, with a correlation of $r^2 = 0.999$.

The hybrid algorithm works by running Merge whenever $m > 0.033n + 8.884$, and running Intersect otherwise. The condition is evaluated on each step of the recursion.

When we modify the algorithm, the cutting point changes. We would like to find the optimal hybrid algorithm. Using the same idea again, we found the straight line that defines the values where Merge is better than the hybrid algorithm. This straight line can be approximated by $m = 0.028n + 32.5$, with $r^2 = 0.992$. Hence, we define the algorithm Hybrid2, which runs Merge whenever $m > 0.028n + 32.5$ and runs Intersect otherwise. Finally, we combined both hybrids, creating a third version where the cutting line between Merge and Intersect is the average between the lines of the hybrids 1 and 2. The resulting straight line is $m = 0.031n + 20.696$. Figure 4 shows the cutting line between the original algorithm and Merge, and the results obtained with the hybrid algorithms. The optimal algorithm would be on theory the Hybrid.i when i tends to infinity, as we are looking for a fixed point algorithm.

We can observe that the hybrid algorithms registered lower times than the original algorithm in the section where the latter is slower than Merge. However, in the other section the original algorithm is faster than the hybrids, due to the fact that in practice we have to evaluate the cutting point in each step of the recursion. Among the hybrid algorithms, we can see that the first one is slightly faster than the second one, and that this one is faster than the third one. An idea to reduce the time in the section that the original algorithm is faster than the hybrids is to create a new hybrid algorithm that runs Merge when it is convenient and that then runs the original algorithm, without evaluating the relation between m and n in order to run Merge. This algorithm shows the same times than Intersect in the section where the latter is better than Merge, combining the advantages of both algorithms in the best way. Figure 5 show the results obtained with this new hybrid algorithm.

5.2 Sequence Lengths with Zipf distribution

As we said before, one of the applications of the algorithm is the search of Web documents, where the number of documents in which a word appears follows a Zipf distribution.

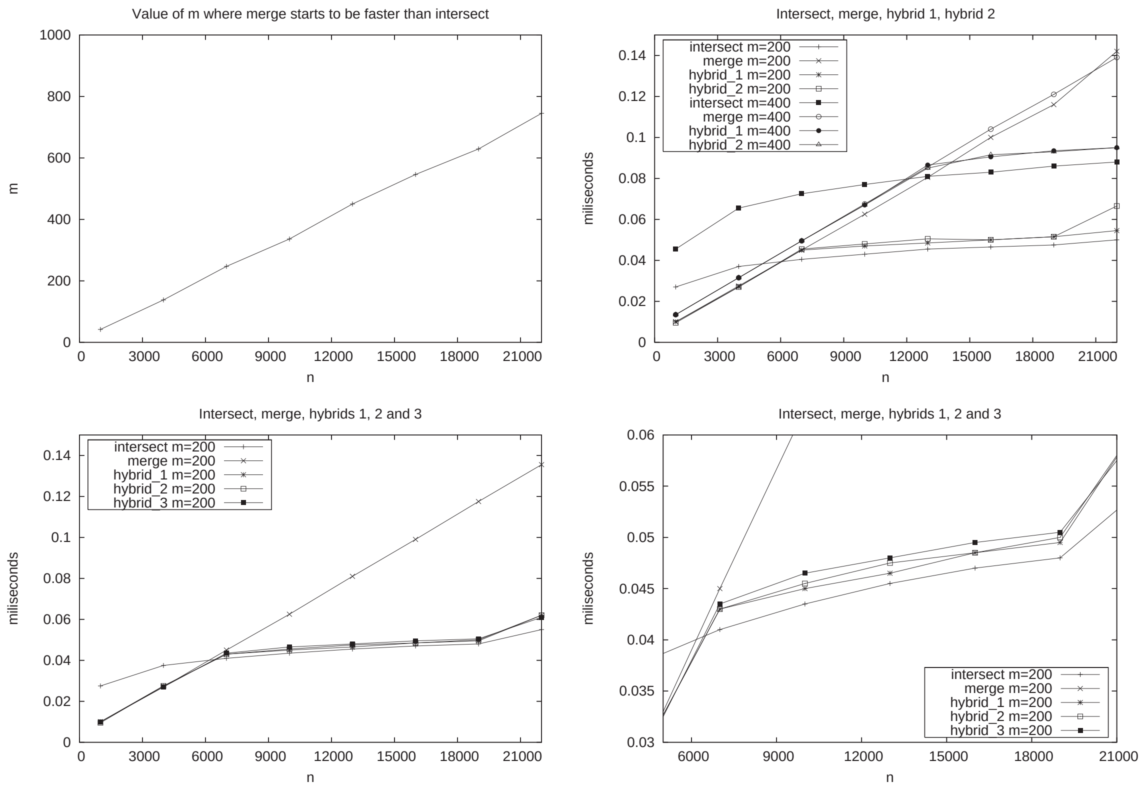

Fig. 4. Up: on the left, value of m from which Merge is faster than Intersect. On the right, a comparison between the original algorithm, Merge and the hybrids 1 and 2 for $m = 200$ y $m = 400$. Down: comparison between Intersect, Merge and the three hybrids for $m = 200$. The plot on the right is a zoom of the one on the left.

It is interesting to study the behavior of the Intersect algorithm depending of the ratio between the lengths of the two sequences when these lengths follow a Zipf distribution and the correlation between both sets is zero (ideal case). For this experiment, we took two random numbers, a and b, uniformly distributed between 0 and 1,000. With these numbers we computed the lengths of the sequences D and Q as $n = K/a^{\alpha}$ y $m = K/b^{\alpha}$, respectively, with $K = 10^9$ and $\alpha = 1.8$ (a typical value for word occurrence distribution in English), making sure that $n > m$. We did 1,000 measurements, using 80 different sequences for each of them, and repeating 1,000 times each run.

Figure 6 shows the times obtained with both algorithms as a function of n/m, in normal scale and logarithmic scale.

We can see that the times of Intersect are lower than the times of Merge when n is much greater than m. When we decrease the ratio between n and m,

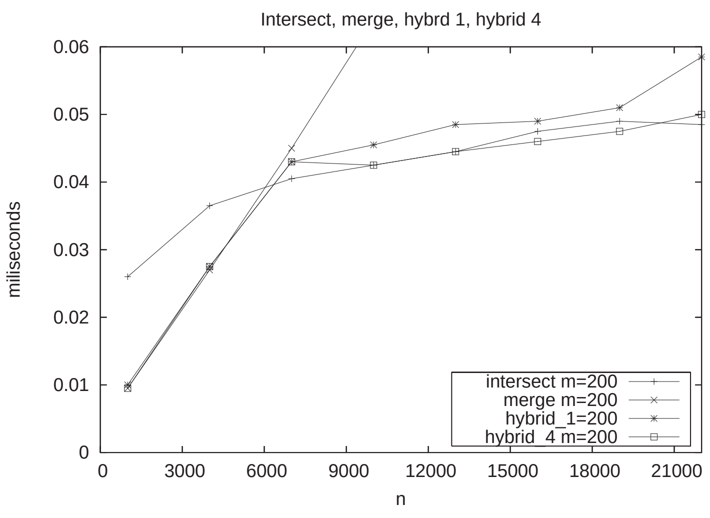

Fig. 5. Experimental results for Intersect, Merge and the hybrids 1 and 4 for different values of n and for $m = 200$.

it is not so clear anymore which of the algorithms is faster. When $n/m < 2$, in most cases the times of Merge are better.

6 Conclusions

In this work we have experimentally studied a simple sorted set intersection algorithm that performs quite well in average and does not inspect all the elements involved. Our experiments showed that Baeza-Yates' algorithm is faster than Merge when one of the sequences is much larger than the other one. This improvement is more evident when n increases. In addition, Baeza-Yates' algorithm surpasses Adaptive [10, 11] for every relation between the sizes of the sequences. The hybrid algorithm that combines Merge and Baeza-Yates' algorithm according to the empiric information obtained, takes advantage of both algorithms and became the most efficient one.

In practice, we do not need to compute the complete result of the intersection of two lists, as most people only look at less than two result pages [6]. Moreover, computing the complete result is too costly if one or more words occur several millions of times as happens in the Web and that is why most search engines use an intersection query as default. Hence, lazy evaluation strategies are used and the results is completed at the user's request.

If we use the straight classical merging algorithm, this naturally obtains first the most relevant Web pages. The same is true for the Adaptive algorithm. For Baeza-Yates' algorithm, it is not so simple, because although we have to process first the left side of the recursive problem, the Web pages obtained do

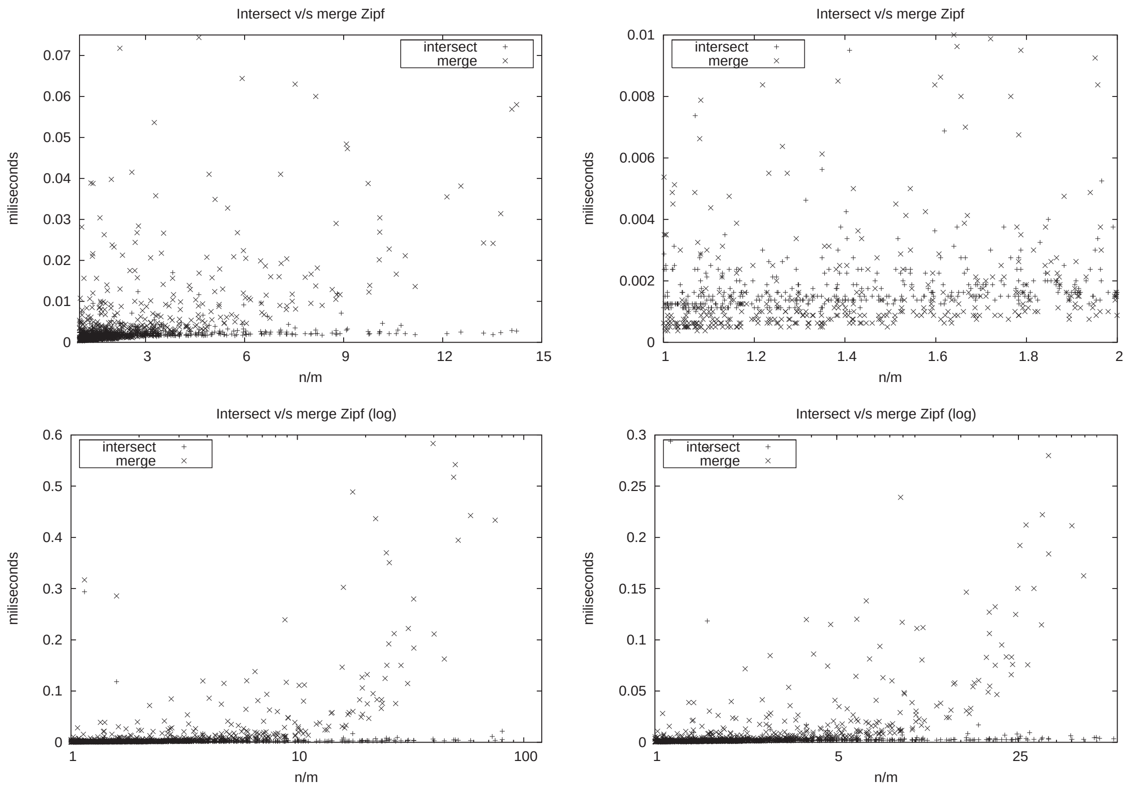

Fig. 6. Up: times for Intersect and Merge as a function of the ratio between the lengths of the sequences when they follow a Zipf distribution. The plot on the right is a zoom of the one on the left. Down: times for Intersect and Merge in logarithmic scale. The plot on the right is a zoom of the one on the left.

not necessarily appear in the correct order. A simple solution is to process the smaller set from left to right doing binary search in the larger set. However this variant is efficient only for small m, achieving a complexity of $O(m \log n)$ comparisons. An optimistic variant can use a prediction on the number of pages in the result and use an intermediate adaptive scheme that divides the smaller sets in non-symmetric parts with a bias to the left side. Hence, it is interesting to study the best way to compute partial results efficiently.

As the correlation between both sets in practice is between 0.2 and 0.6, depending on the Web text used (Zipf distribution with α between 1.6 y 2.0) and the queries (Zipf distribution with a lower value of α, for example 1.4), we would like to extend our experimental results to this case. However, we already saw that in both extremes (correlation 0 or 1), the algorithm on study is competitive.

References

1. R. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, Springer LNCS 3109, pp 400-408, Istanbul, Turkey, July 2004.
2. R.A. Baeza-Yates. *Efficient Text Serching.* PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.
3. Ricardo Baeza-Yates, Phillip G. Bradford, Joseph C. Culberson, and Gregory J.E. Rawlins. The Complexity of Multiple Searching, unpublished manuscript, 1993.
4. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, ACM Press/Addison-Wesley, England, 513 pages, 1999.
5. R. Baeza-Yates, and Felipe Sainte-Jean. A Three Level Search Engine Index bases in Query Log Distribution. SPIRE 2003, Springer LNCS, Manaus, Brazil, October 2003.
6. Ricardo Baeza-Yates. Query Usage Mining in Search Engines. In Web Mining: Applications and Techniques, Anthony Scime, editor. Idea Group, 2004.
7. Ricardo Baeza-Yates, Carlos Hurtado, Marcelo Mendoza and Georges Dupret. Modeling User Search Behavior, LA-WEB 2005, IEEE CS Press, October 2005.
8. Jérémy Barbay and Claire Kenyon. Adaptive Intersection and t-Threshold Problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 390-399, San Francisco, CA, January 2002.
9. S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *7th WWW Conference*, Brisbane, Australia, April 1998.
10. 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.
11. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on Adaptive Set Intersections for Text Retrieval Systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*, LNCS, Springer, Washington, DC, January 2001.
12. Dietz, Paul, Mehlhorn, Kurt, Raman, Rajeev, and Uhrig, Christian; "Lower Bounds for Set Intersection Queries", *Proceedings of the 4th Annual Symposium on Discrete Algorithms*, 194-201, 1993.
13. Dobkin, David and Lipton, Richard; "On the Complexity of Computations Under Varying Sets of Primitives", *Journal of Computer and Systems Sciences*, 18, 86-91, 1979.
14. F.K. Hwang and S. Lin. A Simple algorithm for merging two disjoint linearly ordered lists, *SIAM J. on Computing* 1, pp. 31-39, 1972.
15. Rawlins, Gregory J. E.; *Compared to What?: An Introduction to the Analysis of Algorithms*, Computer Science Press/W. H. Freeman, 1992.