ITAT 2013 Proceedings, CEUR Workshop Proceedings Vol. 1003, pp. 58–64
http://ceur-ws.org/Vol-1003, Series ISSN 1613-0073, © 2013 V. Boža

Experimental Comparison of Set Intersection Algorithms for Inverted Indexing

Vladimír Boža

Department of Computer Science, Faculty of Mathematics, Physics and Informatics Comenius University
Mlynská dolina, 842 48 Bratislava, Slovakia

Abstract: The set intersection problem is one of the main problems in document retrieval. Query consists of two keywords, and for each of keyword we have a sorted set of document IDs containing it. The goal is to retrieve the set of document IDs containing both keywords. We perform an experimental comparison of Galloping search and a new algorithm by Cohen and Porat (LATIN2010), which has a better theoretical time complexity. We show that the new algorithm has often worse performance than the trivial one on real data. We also propose a variant of the Cohen and Porat algorithm with a similar complexity but better empirical performance. Finally, we investigate influence of document ordering on query time.

1 Introduction

In the set intersection problem, we are given a collection of sets $A_1, A_2, \ldots, A_d$. Our goal is to preprocess them and then answer queries of the following type: For given i, j, find the intersection of sets A_i and A_j. This problem appears in various areas. For example, set intersection is needed in conjuctive queries in relational databases, and Ng, Amir, Pevzner [7] use set intersection to match mass spectra against a protein database. However, perhaps the most important application is in document retrieval, where the goal is to maintain data structures over a set of documents. The most typical data structure is the inverted index, which stores for each word the set of documents containing that word. Such an index allows us to easily retrieve documents containing a particular query word. When we want to retrieve documents which contain two or more given words, we can do set intersection of corresponding document sets from the inverted index.

Classical algorithms for set intersection are merging and binary search. Merging identifies common elements by iterating through both sorted lists, as in the final phase of the merge sort algorithm. If we denote the length of the smaller set as m and the larger set as n, then the time complexity of merging is $O(m+n)$, which is good when the lengths of the two sets are almost same. If m is much smaller than n, it is better to search for each element of the smaller set in the larger set by binary search, in $O(m \lg n)$ total time.

There is a better algorithm originally introduced by Bentley and Yao [8], called galloping search with time complexity $O(m \lg(n/m))$. This algorithm has good time complexity when the lengths of the sets are similar and also when the shorter set is much shorter than longer one. We will describe this algorithm in the next section.

All previous algorithms get the two sorted sets on input without any additional preprocessing. However in inverted indexing, sets for all keywords are known in advance, and perhaps some preprocessing of these sets could speed up query processing. In particular, the length of the output can be much smaller than the length of the shorter set, and it would be desirable to have an algorithm, which would not depend linearly on m. The first step in this direction is a recent algorithm by Cohen and Porat [1], which uses linear memory to store all sets and processes each intersection query in time $O(\sqrt{No}+o)$, where o is the length of output and N is the sum of the sizes of all sets. We will denote this algorithm as the fast set intersection algorithm.

However, it is not clear, whether this theoretical improvement is really useful in practice. In this article, we compare the query times of the fast set intersection algorithm and the galloping search on a dataset consisting of a sample of English Wikipedia articles with a set of two-words queries from TREC Terabyte 2006 query stream. We also present a variant of the fast set intersection algorithm with a similar time and memory complexity but better empirical performance.

Previous experimental comparisons of set intersection algorithms [5, 6] consider only different variants of galloping search or other algorithms that do not use set preprocessing. Yan et al. [2] observe that better query times can be achieved via better document ordering. In our experiments, we also compare query times of random document ordering and document ordering based on simple clustering scheme for all tested algorithms.

2 Algorithms

In this section we descibe the three algorithms, which we will compare. We will denote the two input sets as A, B, where $|A| = n$, $|B| = m$, $m \leq n$. We will denote the total number of sets as s and the total size of sets as N.

2.1 Galloping search

The Galloping search ([8]) is a simple modification of the binary search. We try to find each element of B in A; formally for each $B[i]$ we will find index k_i such that $A[k_i] \leq B[i]$ and $A[k_i+1] > B[i]$. We will use several improvements. First, if $j > i$, then $k_j \geq k_i$. This means that when we are searching for k_{i+1}, we need to search only in range $k_i, k_i + 1, \ldots, n$. Secondly before doing binary search we will find the smallest $p \in \{1, 2, 4, 8, \ldots\}$ such that $A[k_i + p] \geq B[i+1]$. This limits the range of the binary search when difference between k_i and k_{i+1} is small. When using this modifications the algorithm achieves time complexity $O(m \lg(n/m))$. Pseudocode of this algorithm is given below:

```
low:=1
for i := 1 to m:
  diff := 1
  while low + diff <= n and A[low + diff] < B[i]:
    diff *= 2
  high := min(n, low + diff)
  k = binary_search(A, low, high)
  if A[k] == B[i]:
    output B[i]
  low = k
```

2.2 The fast set intersection algorithm

We now briefly describe the data structure used by fast set intersection algorithm [1].

We build tree where each node handles some subsets of the original sets. The cost of a node is the sum of the sizes of all the subsets it handles. The root handles all original sets, so it costs N.

Definition 1. *Let d be a node with costs x. A large set in d is one of the $\sqrt{x}$ biggest sets in node d.*

Note the the original article defined large set in a slighly different way. Their large set was a set with at least $\sqrt{x}$ elements. It is clear, that every set with at least $\sqrt{x}$ elements is a large set by our definition.

A set intersection matrix is a matrix that stores for each pair of sets a bit indicating whether their intersection is non-empty. For x sets this matrix needs $O(x^2)$ bits of space and therefore node with cost c needs $O(c)$ bits of memory. For each node of tree we will construct a set intersection matrix for all the large sets in that node.

Now we need to describe how to build the tree. We will use a top-down approach. We will start with root node and in each node we will divide the sets and propagate them to its children. Only large sets are propagated down to the node children. We will call this sets a propagated group. Let d be a node with cost x and G its propagated group. The G costs at most x. Let E be the set of all elements in the sets of G. We will try to split E into two disjoint sets E_1, E_2. For a given set $S \in G$ the child will handle $S \cap E_1$ and the right child will handle $S \cap E_2$. We want the each child to cost at most $x/2$. This is sometimes impossible to achieve. We will fix this by keeping one element of E in d. We add elements to E_1 until adding another element would make the left child cost more than $x/2$. The next element will be kept in d. All other elements will go to E_2 and the right child will cost at most $x/2$.

This tree will have at most $O(\lg N)$ levels. At each level, we need $O(N)$ bits for intersection matrices. This means we need $O(N \lg N)$ bits, which is $O(N)$ in term of words.

During query answering we will start traversing the tree starting in the root node. In each node we will check whether both sets are large. If not, we will answer query using the galloping search. If both sets are large, we will look into intersection matrix. If sets do not have intersection in this node, we stop the traversal in this node. Otherwise, we will propagate the query to the children of that node. We also need to check whether the element kept in the node belongs to the intersection.

It can be shown that the time complexity of a set intersection query is at most $O((\sqrt{No} + o) \lg n)$. It can be also shown that it is never worse than time complexity of the galloping search [1].

Note that the original article used hash tables instead of the galloping search.

2.3 Our set intersection

The set intersection matrix usually contains many ones and only few zeroes. We can use the space better by instead storing intervals in with intersection of two sets is empty. We take $\sqrt{N}$ biggest sets and call them large sets. We will call other sets as small sets. Note that the size of a small set is at most $\sqrt{N}$.

Now we will do a preprocessing for intersections of the large sets.

Definition 2. *Let A, B be two large sets, where $|B| \leq |A|$. The empty interval is a sequence $B[i], B[i+1], \ldots, B[j]$ of elements of set B such that:*

- *For each k such that $i \leq k \leq j$: $B[k] \notin A$.*
- *$i = 1$ or $B[i-1] \in A$.*
- *$j = n$ or $B[j+1] \in A$.*

The size of this interval is $j - i + 1$.

Now we will find and store k largest empty intervals from all intersections of large sets (note that we can store zero, one or more than one intervals for some pairs of sets). Note that if $k > N$, then the smallest stored empty interval has size at most $\sqrt{N}$.

We will answer set a intersection query as follows. If any of the sets is small, we will use the galloping search. This gives us query time $O(m \lg(n/m))$, but since $m \leq \sqrt{N}$, the query time can be written as $O(\sqrt{N} \lg(n/m))$. If both sets are large, then we will again use the galloping search, but we will ignore empty intervals found for the given intersection.

We will show two things about time complexity of the query in our algorithm.

First, if $k = N$, then the query time complexity is bounded by $O(o\sqrt{N})$. The memory complexity in this case is $O(N)$. Secondly, if we put $k = N \lg N$, the average query time complexity of this approach is not worse than the time complexity of the fast set intersection. This is because the number of empty intervals is the same as the number of bits in the set intersection matrices of the fast set intersection algorithm and our empty intervals allows us to skip search for more elements than the zeroes in the set intersection matrices. Thus average query time complexity of our approach is $O((\sqrt{No}+o)\lg n)$. The memory complexity is $O(N\lg N)$, which is little bit higher.

The complexity of our algorithm does not look as promising as complexity of the fast set intersection algorithm, but this algorithm allows time-memory tradeoff. We can store any number of empty intervals as we want. In our experiements we set this number to achieve same memory consumption as fast set intersection algorithm.

3 Document ordering

In the document retrieval we can choose arbitrary IDs for individual documents, and thus influence the ordering of elements in the input sets. There are several proposed heuristics for document ordering; most of them try to order documents for achieving better index compression ([3], [4]). But good document ordering improves query time [2]. This happens because similar document are closer together in the sets and during the galloping search we will make smaller jumps. In our work, we will use the k-scan algorithm [3]. To describe this algorithm, we first need to define similarity of documents.

Definition 3. *The Jaccard similarity of two sets A,B is given by:*

$$J(A,B) = \frac{|A\cap B|}{|A\cup B|}$$

The document is a set of terms. For calculating distance we will only consider $\sqrt{N}$ terms occuring in the largest number of documents (the large sets from the previous sections). The similarity of two documents is the Jaccard similarity of their sets.

The k-scan algorithm tries to find an ordering of documents by partioning them into k clusters. Let d be the number of documents. This algorithm has k iterations. In each iteration, it first picks a cluster center and then chooses among the unassigned documents the $d/k-1$ ones most similar to the cluster center. Also it picks the cluster center for the next iteration, which is the d/k-th most similar document. The cluster center for the first iteration is picked randomly. If we assume that document similarity can be computed in time s, then time complexity of this algorithm is $O(kds)$. In our experiments we use $k=1000$.

4 Experimental setup

We implemented all algorithms in C++ and run our experiment on a computer with Intel i7 920 CPU, 12 GB RAM and Ubuntu Linux. We compiled our code with g++ 4.7.3 using -O3 optimizations.

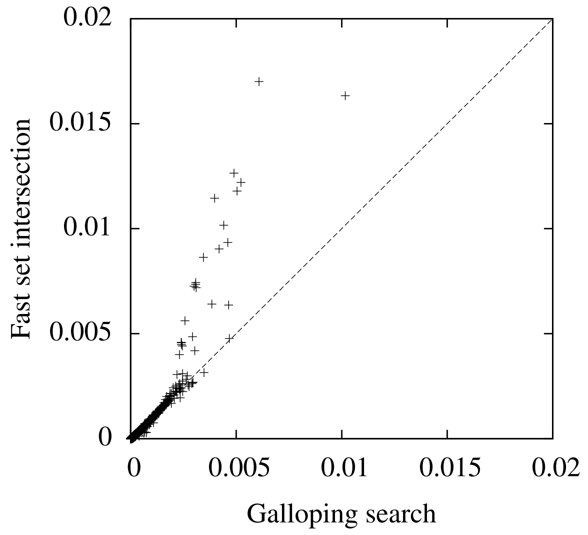

Figure 1: Comparison of query times for galloping search (x-axis) and fast set intersection (y-axis) with random document ordering

4.1 Documents

We took all articles from the English Wikipedia [9] and divided each article into paragraphs. We have used paragraphs instead of documents, because otherwise we would only have a few big documents and the difference between algorithms would be hard to measure. Then we sampled 6.5 millions of paragraphs and took them as documents. The total size of index N (the sum of size of all sets) was 313 millions word-document pairs.

4.2 Queries

We have used query log from TREC Terabyte 2006 query stream [10]. We only consider two word queries from the log. This gives us approximately 14000 queries. We ran each query 100 times and measured the average time in seconds.

5 Experimental results

5.1 Galloping search vs. fast set interserction vs. our algorithm

We will first show comparision between all algorithms using random document order. Results are shown in Figures 1, 2.

As we can see the fast set intersection algorithm introduces significant overhead in query processing time for large queries. Our hypothesis is that this is because this algorithm does not use the caching in an optimal way. On the other hand our set intersection algorithm introduces a small improvement in query processing time. The average improvement is around 14%.

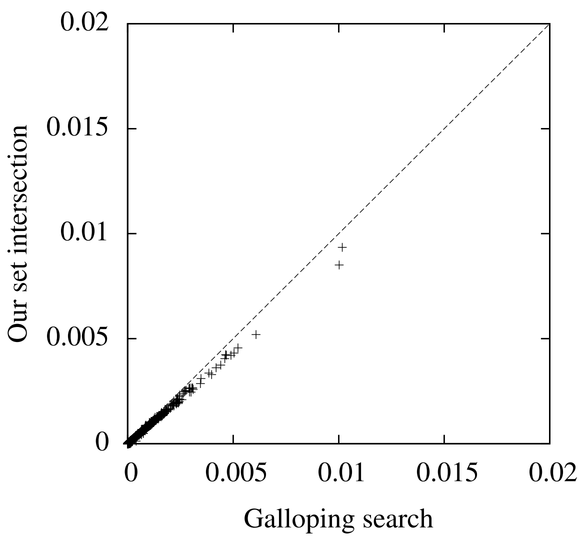

Figure 2: Comparison of query times for galloping search (x-axis) and ours set intersection (y-axis) with random document ordering

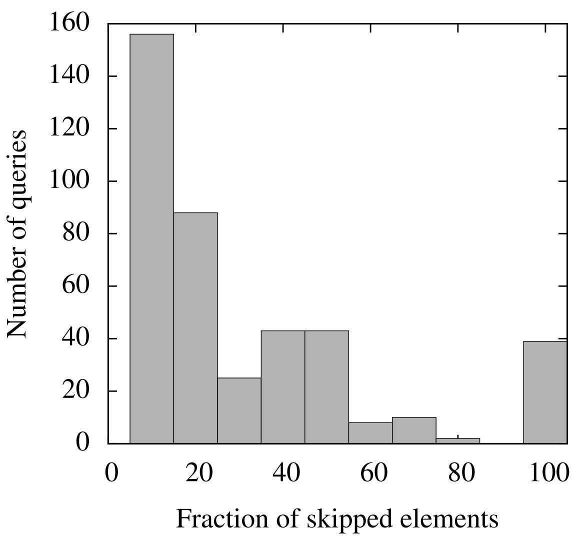

Figure 3: Histogram of fraction of skipped elements from the smaller set in queries for fast intersection algorithm with random document ordering. Zero is ommited due to its big size (12780 queries).

We also explored individual algorithm using more detailed statistics. The set intersection matrices of fast set intersection algorithms contained 15% of zeroes. There are 5400 (39%) queries where both sets are large in root node. Only in 709 of these queries we found zero in the set intersection matrices. We have also measured how many elements of the smaller set we can skip due to zeroes in set intersection matrices. As we can see from the histogram in Figure 3, there are some queries where the output size is zero and all elements are skipped. But overall there are only few queries where we skipped more than half of the smaller set. Most of the time we skip only few percent of the smaller set.

In our algorithm, there are 825 queries where we encounter an empty interval stored for the two sets. Again we measured fraction of skipped elements due to empty intervals with respect to the size of the smaller set and plotted histogram of this fractions (see Figure 4). We see that this histogram looks quite better than the previous one.

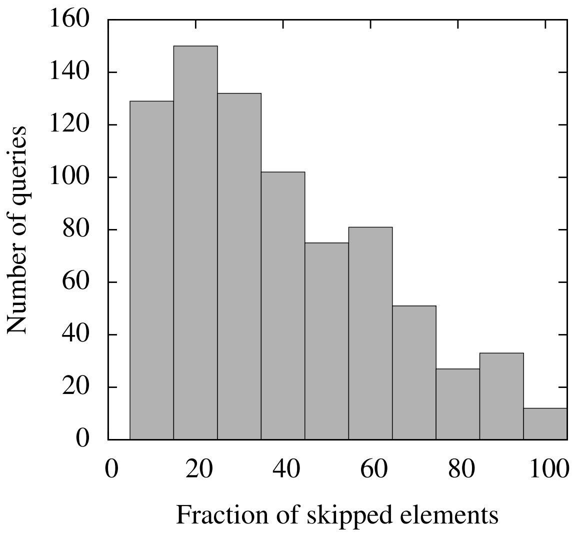

Figure 4: Histogram of fraction of skipped elements from the smaller set in queries for our intersection algorithm with random document ordering. Zero is ommited due to its big size (12400 queries).

5.2 Document ordering effects

The overall effect of document ordering on the query time is shown in Figures 5, 6, 7.

The average improvement of query time for galloping search is 18%, for fast set intersection 27% and for our set intersection 22%.

It is worth noting that the set intersections matrices contain 25% zeroes when using document ordering based on *k*-scans, compared to 15% with random document order. We had 2100 queries which encoutered zero in some set intersection matrix in the fast set intersection algorithm. This approximatelly three times more than when we used random document ordering. Histogram of the fraction of skipped elements is in Figure 8. In this histogram we see the same problems as in random document ordering – the number of queries with 80 – 90 percent fraction is zero. On the other hand, we gained a lot of queries where we eliminated around 10% of work.

In our algorithm, there are 1500 queries where we encounter an empty interval. Histogram of the fraction of skipped elements is in Figure 9. Its shape is similar to histogram when using random document ordering.

Finally, in Figures 10, 11 we see a comparision of running times of algorithms when using document ordering based on *k*-scans. We still see significant slowdown for

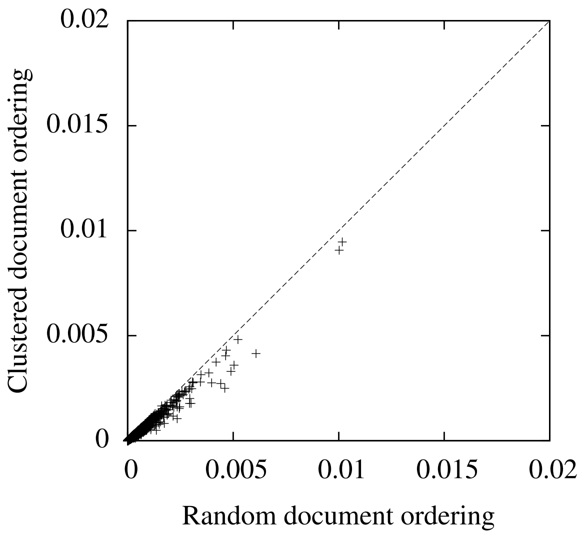

Figure 5: Comparison of query times for galloping search using random document order(x-axis) and document order based on *k*-scan algorithm (y-axis)

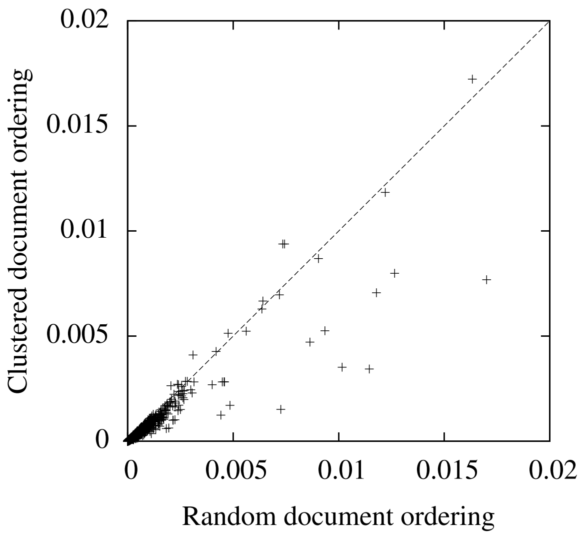

Figure 6: Comparison of query times for fast set intersection using random document order(x-axis) and document order based on *k*-scan algorithm (y-axis)

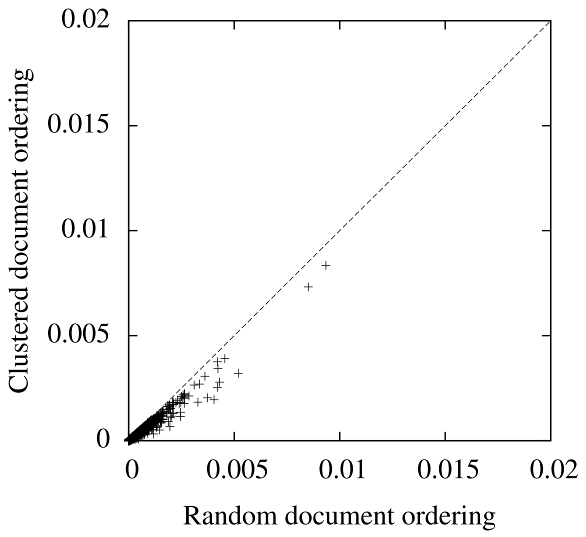

Figure 7: Comparison of query times for our set intersection using random document order(x-axis) and document order based on *k*-scan algorithm (y-axis)

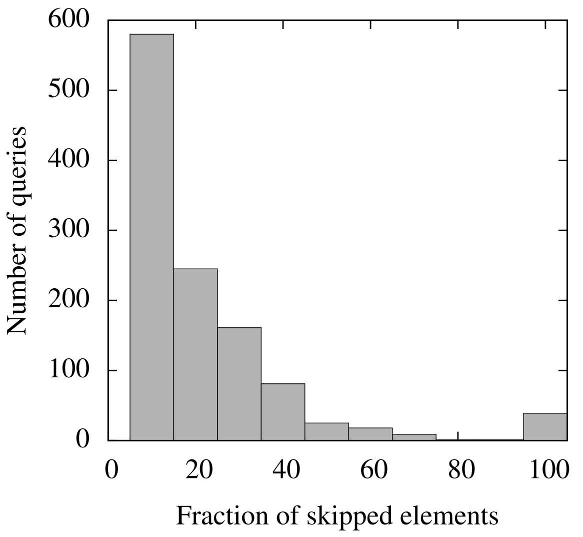

Figure 8: Histogram of fraction of skipped elements from the smaller set in queries for fast intersection algorithm with *k*-scan document ordering. Zero is ommited due to its big size (12390 queries).

fast set intersection. The speedup for our set intersection was 19% which is similar to speedup for random document ordering. Finally, in Figure 12 we see a comparison of galloping search using random document ordering and our algorithm using better document ordering. Combination of these two factors leads to average improvement around 35%.

5.3 Preprocessing time and memory consumption

We now briefly sumarize proprocessing time and memory consumption of our algorithms. Using only inverted index and galloping search took 2 GB of memory and needed 4 minutes for preprocessing. The fast set intersection algorithm required 7 GB of memory and 90 minutes of preprocessing. Our algorithm required 7 GB of memory and 2.5 hours of preprocessing.

6 Conclusion

We examined three different algorithms for set intersection. The experimental result can be summarized as follows:

- Fast set intersection algorithm does not lead to better

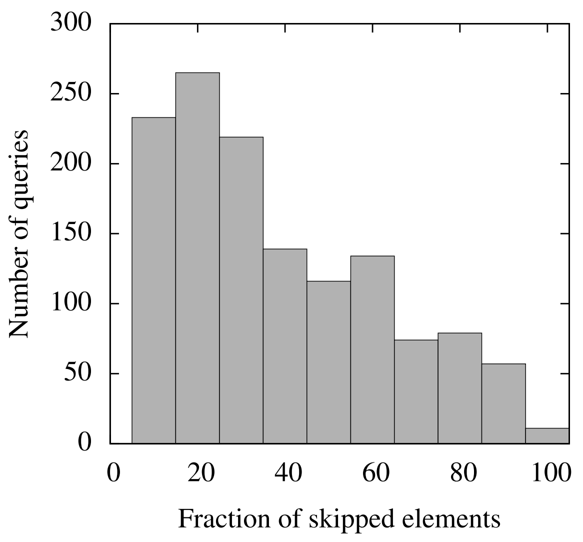

Figure 9: Histogram of fraction of skipped elements from the smaller set in queries for our intersection algorithm with *k*-scan document ordering. Zero is ommited due to its big size (12200 queries).

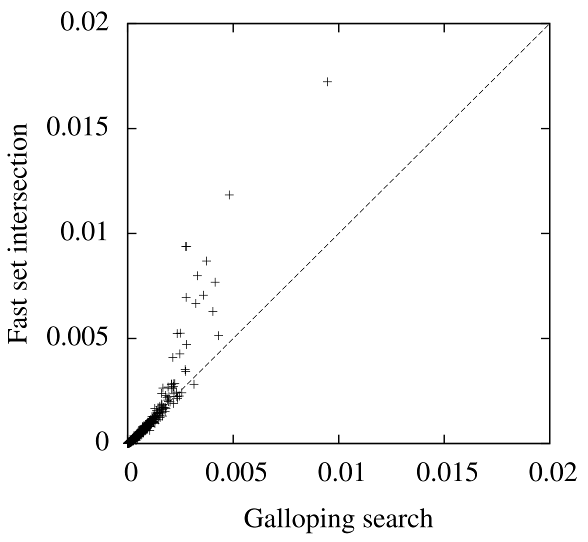

Figure 10: Comparison of query times for galloping search (x-axis) and fast set intersection (y-axis) with document ordering based on *k*-scans

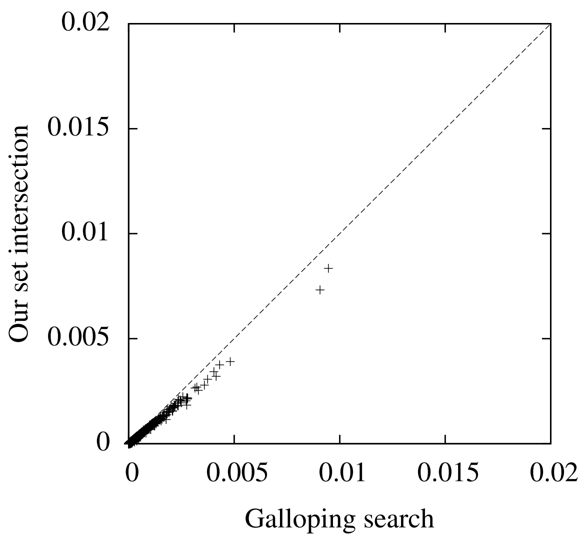

Figure 11: Comparison of query times for galloping search (x-axis) and ours set intersection (y-axis) with document ordering based on *k*-scans

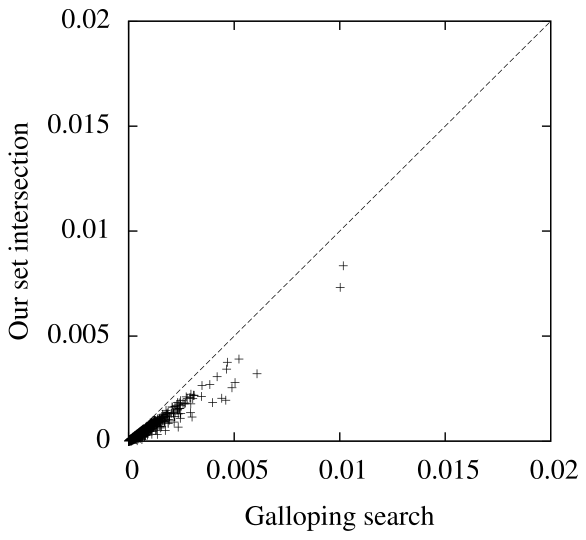

Figure 12: Comparison of query times for galloping search with random document ordering (x-axis) and ours set intersection with document ordering based on *k*-scans (y-axis)

empirical performance on real data.

- We can achieve some speedup using our algorithm but this speed up is not big.
- Our algorithm is slighly better at eliminating useless work than fast set intersection. Fast set intersection algorithm in most cases eliminates less then 10% of work.

It is interesting question whether more careful implementation of fast set intersection algorithm can lead to better query times.

We also investigated effect of document ordering on query times. We showed that better document ordering leads to greater improvement than using a different algorithm.

7 Acknowledgements

This research was supported by VEGA grant 1/1085/12. Author would also like to thank Brona Brejova for many usefull comments.

References

[1] Cohen, Hagai, and Ely Porat: Fast set intersection and two-patterns matching. LATIN 2010: Theoretical Informatics. Springer Berlin Heidelberg, 2010. 234-242.

[2] Yan, Hao, Shuai Ding, and Torsten Suel: Inverted index compression and query processing with optimized document ordering. Proceedings of the 18th international conference on World wide web. ACM, 2009.

[3] F. Silvestri, S. Orlando, and R. Perego.: Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In Proc. of the 27th Annual Int. ACM SIGIR Conf.

[4] Shieh, W. Y., Chen, T. F., Shann, J. J. J., and Chung, C. P. (2003).: Inverted file compression through document identifier reassignment. Information processing & management, 39(1), 117-131.

[5] Culpepper, J. Shane, and Alistair Moffat: Efficient set intersection for inverted indexing. ACM Transactions on Information Systems (TOIS) 29.1 (2010): 1.

[6] Barbay, Jérémy, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger: An experimental investigation of set intersection algorithms for text searching. Journal of Experimental Algorithmics (JEA) 14 (2009): 7.

[7] Ng, Julio, Amihood Amir, and Pavel A. Pevzner.: Blocked pattern matching problem and its applications in proteomics. Research in Computational Molecular Biology. Springer Berlin Heidelberg, 2011.

[8] Jon Louis Bentley and Andrew Chi-chih Yao: An almost optimal algorithm for unbounded searching. Information Processing Letters - IPL , vol. 5, no. 3, pp. 82-87, 1976.

[9] `http://download.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2`

[10] `http://trec.nist.gov/data/terabyte06.html`