A Set Intersection Algorithm Via x-Fast Trie

Bangyu Ye*
National Engineering Laboratory for Information Security Technologies, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, 100091, China.

* Corresponding author. Tel.:+86-10-82546714; email: yebangyu@iie.ac.cn
Manuscript submitted February 9, 2015; accepted May 10, 2015.
doi: 10.17706/jcp.11.2.91-98

Abstract: This paper proposes a simple intersection algorithm for two sorted integer sequences . Our algorithm is designed based on x-fast trie since it provides efficient find and successor operators. We present that our algorithm outperforms skip list based algorithm when one of the sets to be intersected is relatively 'dense' while the other one is (relatively) 'sparse'. Finally, we propose some possible approaches which may optimize our algorithm further.

Key words: Set intersection, algorithm, x-fast trie.

1. Introduction

Fast set intersection is a key operation in the context of several fields when it comes to big data era [1], [2]. For example, modern search engines use the set intersection for inverted posting list which is a standard data structure in information retrieval to return relevant documents. So it has been studied in many domains and fields [3]-[8]. One of the most typical situations is Boolean query which is required to retrieval the documents that contains all the terms in the query. Besides, set intersection is naturally a key component of calculating the Jaccard coefficient of two sets, which is defined by the fraction of size of intersection and union.

In this paper, we propose a new algorithm via x-fast trie which is a kind of advanced data structure. We focus on intersection for two sets due to its fundamental status. Experiments show that when one of the sets to be intersected is relatively 'dense' while the other one is (relatively) 'sparse', our algorithm outperforms skip list based algorithm.

2. Related Work and Motivation

2.1. Related Work

Suppose that the two sets intersected are denoted as S and T, with size n and m, respectively ($m<n$).

Most of the algorithms suppose that the elements in the set are stored in arrays. One obvious way is to lookup each element of one set in the other one. There are many possible search algorithms can be used here [9]. If the array is ordered, we can run m times binary search over the larger set whose size is n, with $O(m\lg n)$ time cost. When $m=O(n/\lg n)$, it is better than linear merge algorithm which takes $O(m+n)$.

Ref. [10]-[13] propose some adaptive algorithms which use the total number of comparisons as measure to evaluate the performance. However, experiment result in [14] shows that such kind of algorithms does not always reduce the time cost in practice.

There are many algorithms that take use of advanced data structures rather than array to accelerate the

processing of set intersection [15]-[17]. One classical approach is skip-list based algorithm [18], [19]. With the help of skip pointers, skip–list based intersection algorithm can avoid some unnecessary comparisons. Apparently, it may not widely used in reality due to the required space-overhead.

Another intuitive approach is to store the larger set whose size is m in a hash table, and lookup all elements of smaller set in the hash table. It takes $O(n)$ time. However, this approach performs poorly in practice. In order to get a better performance, a new algorithm is proposed in [20]. Firstly, it uses a hash function to map the elements in two sets into two relatively smaller presentations, denoted as Hash(S) and Hash(T) respectively. Secondly, the Hash(S) and Hash(T) will be intersected. Lastly, the false positives will be removed. In terms of intersection for two sets, this algorithm takes in expected time $O((m+n)(\log w)^2/w+2r)$, in which r stands for the size of the intersection and w denotes the length of machine word. Obviously, this bound relies on a large value of w. In [21], set is presented by linear-space data structure which allows to compute the intersection in $O((m+n)/\mathrm{sqrt}(w)+2r)$.This algorithm performs better in practice.

Experiments in [22], [23] compare several intersection algorithms and show that the complexity of intersections relies heavily on the distributions of the elements in the sets.

Ref. [24] focus on the situation where multi-cores CPU are available and does study how to improve the performance of list intersection.

2.2. Motivation

Suppose that the two sets to be intersected are S and T whose size are n and m respectively. Our approach leverages a key observation: Currently, we are comparing $S[i]$ and $T[j]$ and $S[i]<T[j]$. If We can determine that $S[i+1],S[i+2]\ldots S[m]$ are all less than $T[j]$, we can skip the elements from $S[i+1]$ to $S[m]$ and continue comparing from $S[m+1]$ and $T[j]$.

For example:

$$S=\{1,2,3,4,5,6\ldots 10000000\},\ T=\{10000,20000,30000\ldots 10000000\}$$

We start comparing 1 in S and 10000 in T. 1<10000 means that firstly we search 10000 in S to check whether 10000 is one of the elements in intersection or not. After that, we try to locate the next element should be compared in S. It is 10001.Hence the elements from [1, 9999] in S are all skipped since they are absolutely not the answers. We implement search (find) and locate (successor) operations efficiently via x-fast trie and hope that one locate operation could skip as many elements as possible although it relies heavily on the distributions of the two sets.

3. Our Methods

3.1. Notation

Suppose that the two sets to be intersected are denoted as S and T, with size n and m, respectively ($m<n$). The elements in the two sets are integers and all belong to $[1, U]$($S[1]<S[2]<S[3]<\ldots<S[n]$; $T[1]<T[2]<T[3]<\ldots<T[m]$). The key operators of our algorithms are Find and Successor:

Find(S, x): lookup x in S to check S contains x or not. If it does, return true; otherwise return false.

Successor(S, x): return the successor of x in S. When x equals to the maximum of S, it just returns + inf.

3.2. Data Structure

X-fast trie [25] was proposed by Dan Willard in 1982 [26]. It is a bitwise trie. If an internal node has no left child, it stores a descendant pointer to the smallest leaf in its right subtree. Likewise, if it has no right child, it stores a descendant pointer to the largest leaf in its left subtree as shown in Fig. 1 [27]. Each leaf node contains a key, a pointer to its predecessor and a pointer to its successor. All leaf nodes are structed by a double linked list. All nodes are stored in hash tables Table[1],Table[2]…Table[h] where h is the height of the

tree and $h=O(\lg U)$. Each level corresponds to one hash table. Typically, hash tables are implemented by cuckoo hashing or dynamic perfect hashing [28], [29].

The space usage of an x-fast trie containing n integers is $O(n\lg U)$, since each element has a root-to-leaf path of length $O(\lg U)$.

It takes $O(1)$ to find a key and the time for finding the successor of an element takes $O(\lg\lg U)$ in an x-fast trie.

Find:

In order to check whether an x-fast trie contains key x or not, we can look up the Table[1] which corresponds to the leaf nodes., this takes $O(1)$ time.

Successor:

In order to find the successor of x, firstly, we run a binary search algorithm to locate the node which has the longest common prefix with x, denoted as lcp(x). Once we find the lcp(x), it is either an internal node or leaf node. If it is an internal node , we can get the successor of x via its descendant pointer with constant time. If it is a leaf node, which means that the trie contains x, we can get the successor of x via the pointer which pointed to its successor, which also takes constant time. Since the height of an x-fast trie is at most $O(\lg U)$, the total time for finding the successor of an element takes $O(\lg\lg U)$.

Suppose that at one time, the content in the hash tables of an x-fast trie which contains elements which range from 1-256 are:

Table[1]:00101011，01010011，10110101
Table[2]: 0010101, 0101001, 1011010
Table[3]: 001010, 010100, 101101
Table[4]:00101,01010,10110
Table[5]:0010,0101,1011
Table[6]:001,010,101
Table[7]:00,01,10
Table[8]:0,1

We can know that the x-fast trie contains 43(00101011b), 83(1010011b) and 181(10110101b). In order to get the successor of 49, firstly we find the node which has the longest common prefix of 00110001. We run binary search over the 8 hash tables and get 001 in Table[6]. After that, we can find the successor via its right pointer which pointed to the successor node at the leaf level.

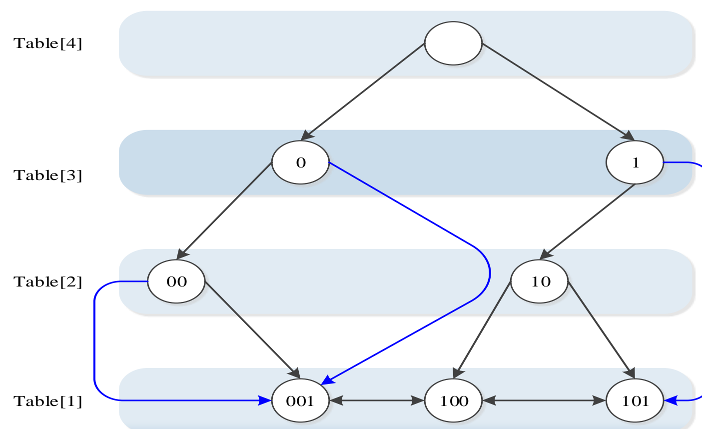

Fig. 1. An x-fast trie containing the integers 1(001b), 4(100b) and 5(101b).

3.3. Algorithm

We design an algorithm based on x- fast trie, as shown in Table 1. The elements in S and T are stored at two x- fast tries.

We start comparing the elements of S and T from $S[1]$ and $T[1]$. When x or y equals +inf, it means that there are no candidates any more. So the loop finished.

Since the loop runs $\min(n, m)$ times at most, X Intersection algorithm needs $O(\min(m, n)\lg\lg U)$ time.

Table 1. X Intersection Algorithm

```
X Intersection Algorithm
Input: Two sets S and T. Both are presented via x-fast trie
Output: The intersection of S and T
x= S[1]     y=L[1]
While(x!=+inf   &&   y!=+inf) do
     if(x==y)
          add x to the intersection set
          x=Successor(S, x)
          y=Successor(T, y)
     else if(x>y)
          if(Find(T, x))
            add x to the intersection set
          y=Successor(T, x)
     else
          if(Find(S, y))
            add y to the intersection set
          x=Successor(S, y)
```

4. Experiment

We evaluate the performance of our algorithm and skip-list based intersection algorithm which also aimed to skip the elements that absolutely can not be the candidates efficiently in this paper.

Implementation: For each Algorithm, we try our best to optimize its performance. The skip list based algorithm is implemented according to [9]. We firstly load the data into memory, store the elements into x-fast trie and skip list respectively. After that we run the core code for intersection algorithm and start counting time. The codes are implemented by C++ in Visual Studio 2013 with no optimization options enabled.

4.1. Experiments on Synthetic Data

The elements in two sets are generated from a universe in which the elements range from 1 to 10,000,000. In order to make set S to be 'dense', S is selected from those candidates as below:

$$S1=\{1, 2, 3, 4, 5, 6, \dots 10000000\}$$

$$S2=\{2, 3, 4, 5, 6, 7, \dots 10000000\}$$

…

$S100$={100, 101, 102, 103, 104, 105, … 10000000}

We varies the gap of *T* from 10 to 1,000,000, which makes *T* more and more 'sparse'. In other words, *T* is one of those candidates as below:

$T1$={10, 20, 30, 40, 50, … 10000000}

$T100$={100, 200, 300, …, 10000000}

…

$T1000000$={1000000, 2000000, …, 10000000}

In *j*th (*j*=1, 2, 3 … 100) iteration, we fix *S* to be *Sj* and compute intersection of *Sj* and *T*1, *T*100 … *T*1000000 respectively.

After 100 iteration (responds to *S*1, *S*2 … S100) are completed, the average times are reported in Fig. 2.

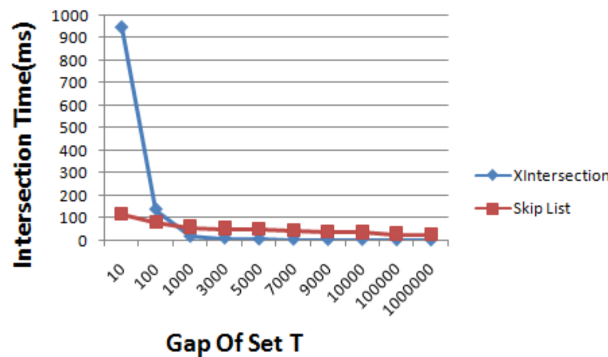

Fig. 2. Experimental results on synthetic data.

4.2. Experiments on Real Data

In experiments on real data, we use a set of 945,147 Chinese news documents which are crawled from several news site including news.163.com, news.sina.com, and news.fenghuang.com .We extract the plain text using Tika and Jsoup ,and construct the posting lists in memory both by x-fast trie and skip list. All terms except for stop words are reserved.

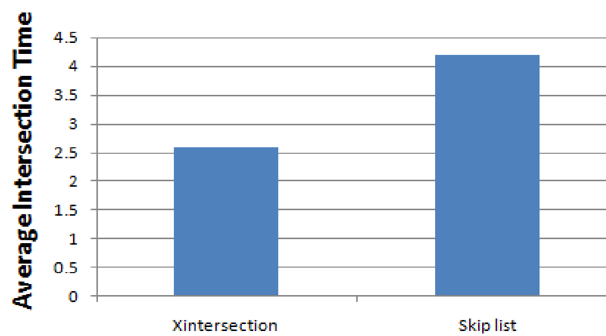

Fig. 3. Experimental results on real data.

We are more interested in the situations that one of the sets to be intersected is relatively small while the other one is large, since the larger set which contains more elements tends to be dense. One successor operations on this set will skip (expectedly) many elements in order to avoid the unnecessary comparisons. In other words, we use the size of sets as a kind of heuristic information. The size of the larger set range from 400,000 to 760,000 while the smaller one range from 3,000 to 7,000. The average ratioof size of two sets is

0.01.

We run 50000 cases in total. In each case, our algorithm outperforms skip-list based algorithm. The average time of XIntersection takes is 2.6ms and the average time of skip-list based algorithm takes is 4.2ms as reported in Fig. 3.

5. Future Work and Discussions

5.1. Formal Definition and Description

The performance of our algorithm relies heavily on the distributions of elements in the two sets to be intersected. Intuitively, when one of the set whose elements are compacted and consecutive is relatively 'dense' and the other one is 'spase', our algorithm may work better. We just used the size of sets as a kind of heuristic information while did not give the rigorous definitions of 'density' and 'spasity'. We leave this as a significant part of our future work.

5.2. Doc Id Reordering

In our experiments on real data, the doc ids are ordered in terms of urls of documents. The documents of same topic will be assigned to consecutive and compact ids. This ordering approach tends to give tight clustering throughout the lists. Indeed, there are many ordering methods [30]-[32] such as random assignment can be applied here. Whether the Doc id reordering may improve the performance of our algorithm needs more experiments and analysis.

5.3. Space Usage

In our experiments on real data, the space usage of our algorithm is almost 10 times than skip-list based algorithm. Since an x-fast trie takes $O(n \lg U)$ spaces for n elements which range from 1-U, it may not very useful in reality. To reduce the overhead of space usage, one possible approach is to design an algorithm based on y-fast trie [33] which required $O(n)$ space usage for storing n elements which range from 1-U.The total time remains $\min(m, n)\lg\lg U$.

5.4. Hashing Function

In our implementation, the hashing function we use is murmurhash2 which is a famous and typical choice when a hashing function is necessary. Note that the content in the hash table is nodes of x-fast trie. Obviously, other hashing functions may be available here. Whether changing the hash function from murmurhash2 to another one will bring a better result still remains a question. In order to get a better performance of x-fast trie, it is necessary to take a better consideration of hashing functions.

5.5. A Hybrid Algorithm

In our experiments, our algorithm outperforms skip-list based algorithm when the sparsity of the two sets to be intersected are quite different. Since the state of the art of this situation is look up algorithm based hashing table, whether our algorithm outperforms it remains a question. It is possible for us to make a hybrid approach which consists of a lot of algorithm to adjust to the different datasets. It means that we call different routine when we meet different algorithm. In fact, we can generate a framework of many algorithm based x-fast trie.

6. Conclusions

In this paper, we propose an intersection algorithm via x-fast trie. We have shown that when one of the sets to be intersected is relatively 'dense' and the other one is 'sparse', our algorithm outperforms skip-list based algorithm due to the efficient successor operator which takes $O(\lg\lg U)$ time .

Acknowledgment

The research work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06030200 and the National Key Technology R&D Program under Grant No. 2012BAH46B03.

References

[1] Brutlag, J. *Speed Matters for Google Web Search*. From: http://code.google.com/speed/files/delayexp.pdf

[2] *We Knew Web Was Big*. From: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

[3] Ding, S., & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 993-1002). ACM.

[4] Ding, S., Attenberg, J., & Suel, T. (2010). Scalable techniques for document identifier assignment ininverted indexes. *Proceedings of the 19th International Conference on World Wide Web* (pp. 311-320). ACM.

[5] Sanders, P., & Transier, F. (2007). Intersection in integer inverted indices. *Proceedings of ALENEX*: *Vol. 7* (pp. 71-83).

[6] Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using at wo-level retrieval process. *Proceedings of The Twelfth International Conference on Information and Knowledge Management* (pp. 426-434). ACM.

[7] Chiniforooshan, E., Farzan, A., & Mirzazadeh, M. (2001). Worst case optimal union-intersection expression evaluation. *Proceedings of ALENEX*.

[8] Tsirogiannis, D., Guha, S., & Koudas, N. (2009). Improving the performance of list intersection. *Proceedings of the VLDB Endowment*, *2(1)*, 838-849.

[9] Bentley, J. L., & Yao, A. C. C. (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters*, *5(3)*, 82-87.

[10] Baeza-Yates, R. (2004). A fast set intersection algorithm for sorted sequences. *Proceedings of Combinatorial Pattern Matching* (pp. 400-408). Springer Berlin Heidelberg..

[11] Barbay, J., & Kenyon, C. (2002). Adaptive intersection and t-threshold problems. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 390-399). Society for Industrial and Applied Mathematics.

[12] Barbay, J., López-Ortiz, A., & Lu, T. (2006). Faster adaptive set intersections for text searching. *Experimental Algorithms*, pp. 146-157.

[13] Demaine, E. D., López-Ortiz, A., & Munro, J. I. (2000). Adaptive set intersections, unions, and differences. *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms.*

[14] Demaine, E. D., López-Ortiz, A., & Munro, J. I. (2001). Experiments on adaptive set intersections for text retrieval systems. *Algorithm Engineering and Experimentation*, 91-104.

[15] Blelloch, G. E., & Reid-Miller, M. (1998). Fast set operations using treaps. *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (pp. 16-26).

[16] Brown, M. R., & Tarjan, R. E. (1979). A fast merging algorithm. *Journal of the ACM*, *26(2)*, 211-226.

[17] Adelson V. M., & Landis, E. M. (1962). An algorithm for the organization of information. Doklady Akademii Nauk SSSR.

[18] Pugh, W. (1990). *A skip List Cookbook* (Technical ReportUMIACS-TR-89-72.1). University of Maryland.

[19] Pugh, W. (1990). Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, *33(6)*, 668-676.

[20] Bille, P., Pagh, A., & Pagh, R. (2007). Fast evaluation of union-intersection expressions. *Algorithms and*

Computation, 739-750.

[21] Ding, B., & König, A. C. (2011). Fast set intersection in memory. *Proceedings of the VLDB Endowment, 4(4)*, 255-266.

[22] Baeza-Yates, R., & Salinger, A. (2005). Experimental analysis of a fast intersection algorithm for sorted sequences. *String Processing and Information Retrieval*, 13-24.

[23] Barbay, J., López-Ortiz, A., Lu, T., & Salinger, A. (2010). An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*.

[24] Tatikonda, S., Junqueira, F., Cambazoglu, B. B., & Plachouras, V. (2009). On efficient posting list intersection with multicore processors. *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information retrieval* (pp. 738-739).

[25] *X-Fast and y-Fast Tries*. From: http://web.stanford.edu/class/cs166/lectures/15/Small15.pdf

[26] Willard, D. E. (1983). Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters, 17(2)*, 81-84.

[27] *X-Fast Trie*. From: http://en.wikipedia.org/wiki/X-fast_trie

[28] Pagh, R., & Rodler, F. F. (2001). "Cuckoo hashing". Algorithms — ESA 2001. *Lecture Notes in Computer Science*.

[29] Which hashing algorithm is best for uniqueness and speed? From: http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

[30] Long, X., & Suel, T. (2003). Optimized query execution in large search engines with global page ordering. *Proceedings of the 29th International Conference on Very Large Data Bases: Vol. 29* (pp. 129-140).

[31] Shi, L., & Wang, B. (2012). Yet another sorting-based solution to the reassignment of document identifiers. *Information Retrieval Technology* (pp. 238-249). Springer Berlin Heidelberg.

[32] Silvestri, F., Orlando, S., & Perego, R. (2004). Assigning identifiers to documents to enhance the clustering property of fulltext indexes. *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 305-312). ACM.

[33] Y-Fast Trie. From: http://en.wikipedia.org/wiki/Y-fast_trie

Bangyu Ye was born in Fujian, China, in 1991. He received his B.S. degree in Quanzhou Normal University at 2012. Currently, He is a postgraduate student in the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include data mining , machine learning and large scale data storage.