MIDDLE EAST TECHNICAL UNIVERSITY, CENG 776

Efficient set intersection for inverted indexing paper report

Ozgur GUNDOGAN
Burak Sefa SERT

January 22, 2018

1 INTRODUCTION

A conjunctive query q is equivalent to a |q|-way intersection which uses ordered sets of integers, where each set represents the documents containing one of the terms, and each integer in each set is an ordinal document identifier. Therefore, there is a tension between the way in which the data is represented, and the ways in which it is to be manipulated. In this paper, this tension tradeoffs are explored by investigating different intersection techniques. Also, author of this paper proposes a new simple hybrid method to make intersection computation faster.

2 ALGORITHMS FOR EFFICIENT F-SEARCH

To intersect two or more lists , we need to search the items in other lists. Also, finger search (F-Search) step is the an important part of an intersection process as computationally. In this part , we will discover different type of f-search algorithms.

Although binary search is not so efficient search algorithm, it is used in other search algorithms. That's why it is needed to be mentioned here with an example. To examplify binary search, let's think about two ordered sets S and T , with |S| = s and |T| = t, and $s \leq t$. Binary search over t elements requires $1 + \log t$ comparisons. In particular, binary search is the optimal approach when s = 1.

There are also other searching methods linear search, interpolation search, Fibonacci search, exponential search and Golomb search. The desirable characteristic of these alternatives is that the search cost grows as a function of the distance traversed, rather than the size of the array. For example, linear search requires O(d) time to move the finger by d items; exponential search requires $O(\log d)$ time.

In situations when $1 \ll n_1 \ll n_2$, use of exponential search is considerable benefit. In an exponential search, probes into T are made at exponentially increasing rank distance from the current location, until a value greater than the search key is encountered. A binary search is then carried out within the identified subrange. In this approach each F-SEARCH call requires $1 + 2 * \log d$ comparisons, where d is the difference between the rank of the finger's previous position and the new rank of the finger pointer.

Golomb searching also can be used.Progressively, search proceeds is similar to exponential search, but with a fixed forwards step of b items used at each iteration, not exponentially increasing. Once overshoot has been achieved, a binary search takes place over the (lastly) b items that have been identified . When searching through a set of size n_2 for the elements of a set of size n_1, the correct value for the step b is 0.69(n2/n1), with a total search cost that is again proportional to O(n_1 + n_1 log(n_2/n_1)).
Golomb and binary search are implemented in python. It can be seen here.

3 INTERSECTION METHODS

3.1 BINARY INTERSECTION OF ORDERED SETS

In binary Intersection of ordered sets, we intersect two lists one by one. Progressively, each element of the smaller set, S, is tested against the larger set, T , and retained if it is present. The search retains state as it proceeds, with the eliminator element, x, stepped through the elements of S; and the F SEARCH (finger search) operation used in T to leapfrog over whole subsequences, pausing only at one corresponding value in T for each item in S.

3.2 SMALL VERSUS SMALL

When more than two sets are being intersected, the simplest approach is to iteratively apply the standard two-set intersection method using as a sequence of pairwise operations.Small versus small (svs) approach is based on this idea. Progressively, the smallest set is identified, and then that set is intersected with each of the others, in increasing order of size. The svs method is simple and effective, and benefits from the spatial locality inherent from processing the sets two at a time. Even so, each different F-SEARCH implementation gives rise to a different svs computation.

3.3 ADAPTIVE HOLISTIC INTERSECTION

The alternative to the svs approach is to combine all of the sets using a single sweep through them all. The resultant holistic algorithms offer the possibility of being adaptive to the particular data arrangement present. The simplest holistic approach is to treat each item in the smallest set as an eliminator, and search for it in each of the remaining sets. Conceptually, this method is identical to an interleaved version of svs.
In adaptive holistic intersection (adp), the sets are initially monotonically increasing in size. At each iteration, the eliminator is the next remaining item from the set with the fewest remaining elements. If a mismatch occurs before all sets have been examined, the sets are reordered based on the number of unexamined items remaining in each set, and the successor from the smallest remaining subset becomes the new eliminator. This approach reduces the number of item-to-item comparisons expected to be required, but at the possibly non-trivial cost of reordering the |q| lists at each iteration of the main loop.

3.4 SEQUENTIAL HOLISTIC INTERSECTION

In sequential holistic intersection (seq) , algorithm uses as the next eliminator the element that caused the previous eliminator to be discarded, and continues the strict rotation among the sets from that point. Only when an eliminator value is found in all the sets âĂŞ and hence is part of the intersection's output âĂŞ is a new eliminator chosen from the smallest set. This approach has the advantage that the sets do not need to be reordered, while still allowing all of the sets to provide eliminators. However, this method suffers from a practical disadvantage: more F-SEARCH operations are likely to accrue when the eliminator is drawn from a populous set than when it is drawn from one of the sparse sets in the intersection.

3.5 Max Successor Intersection

Holistic methods may have a memory access pattern that is less localized than do svs methods, because all of the sets are processed concurrently. To diminish this risk, author of the paper propose a further alternative. The eliminator is initially drawn from the smallest set. When a mismatch occurs, the next eliminator is the larger of the mismatched value and the successor from the smallest set. Processing starts in S_2 if the eliminator is again taken from S_1, otherwise processing begins in S_1. The intuition behind this approach is two-fold. The first is that, without any other information, the best eliminator will be in the smallest set. The second intuition is that, having discovered a bigger than anticipated jump in one of the sets, that value should be tested against the first set, to see if additional items can be discarded.

4 DATASET

The dataset we have used includes more than 500,000 words and 41,000,000 posting list entry. It is quite big, therefore instead of importing the whole list into memory we have lazy loaded the posting list data as needed. Apart from that, query that we have used is extracted from as title of query topics, with punctuation omitted. The remaining part is split by whitespace and each word in query is AND'ed.

5 EXPERIMENTS AND RESULTS

Throughout this project, we have implemented four methods which are small versus small, adaptive holistic intersection, sequential holistic intersection, max successor intersection. As search function, we used golomb search in max successor intersection algorithm and binary search in the rest of algorithms. Since our dataset consists queries upto 5 words in a query, we needed to concatenate queries randomly to generate queries more than 5 words. Thanks to that, we compared methods' efficiencies with respect to query length in a better way. As can be seen in Figure 5.1, small versus small is the worst algorithm and adaptive holistic intersection algorithm is the best one in average. It can be said that the performance of the algorithms are highly query dependent if we compare query length 4 and 5. Result in query length 4 and 5 show us the adaptive holistic intersection algorithm is better than max successor when query length is 4 but worse than max successor when query length is 5. Also, if we examine result of any of algorithms in 5.1, we can see that algorithms performance do not decrease when query length increases as expected. Actually it makes sense if we think about one word query which has a long posting list and three words query which all words have a short posting list.

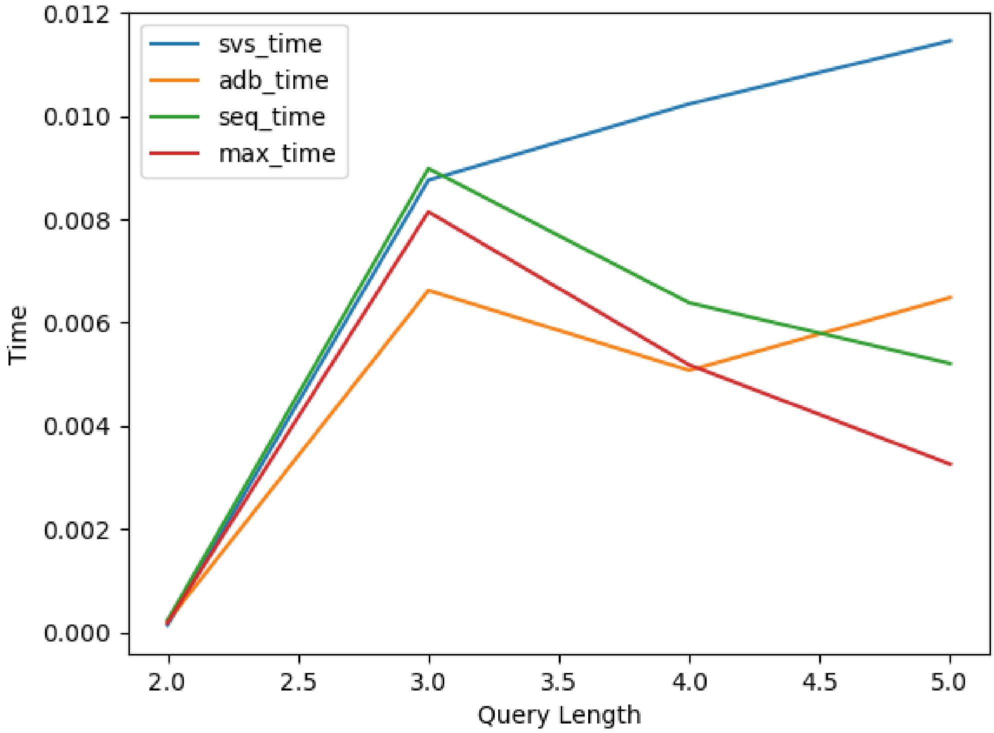

Figure 5.1: Original queries upto 5 words in a query

All observations in 5.1 which mentioned above are still valid for 5.2. In this graph, we easily observe that some queries consume so much time with respect to others.

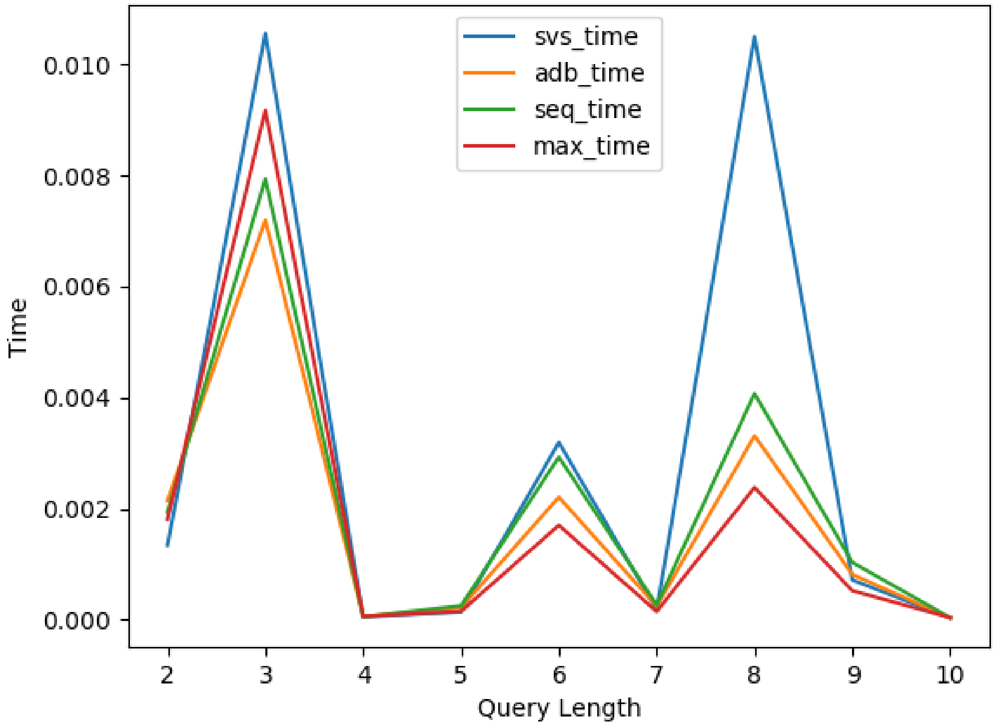

Figure 5.2: Concatenated queries upto 10 words in a query

6 CONCLUSION

By using the results for our dataset, we can not say that one of the algorithms is better than others. Adaptive holistic intersection and max successor intersection algorithms work slightly better than others. However, performance of these two algorithms can not be comparable. We can suggest using a combined version of these two algorithms with respect to queries which they work well.

7 SOURCE CODE

All source code can be found under Github Repository.

7.1 INTERSECTION ALGORITHMS IMPLEMENTATION

```
from Searchs import *

BinaryIntersection = 0
SVSIntersection = 1
ADBIntersection = 2
SEQIntersection = 3
MAXIntersection = 4


class Intersections():
    def __init__(self, type):
        self.type = type

    def intersect(self, list_of_input_lists):
        if (self.type == BinaryIntersection):
            return self.binary_intersect(list_of_input_lists)
        elif (self.type == SVSIntersection):
            return self.svs_intersect(list_of_input_lists)
        elif (self.type == ADBIntersection):
            return self.adp_intersect(list_of_input_lists)
        elif (self.type == SEQIntersection):
            return self.seq_intersect(list_of_input_lists)
        elif (self.type == MAXIntersection):
            return self.max_intersect(list_of_input_lists)

    def binary_intersect(self, list_of_input_lists, sorrt=True ,verbose=False):

        assert len(list_of_input_lists) == 2

        if(sorrt):
            # sort list wrt its length
            listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)
        else:
            listsAreSortedWRTLength = list_of_input_lists

        intersectedArray = []

        # split short and long list
        shortList, longList = listsAreSortedWRTLength[0], \
                              listsAreSortedWRTLength[1]


        for ind in range(len(shortList)):
```

```
            # calculate golomb parameter

            x = shortList[ind]
            golombParameter = int(len(longList) / (len(shortList)-ind))

            # if(golombParameter > 40):
            #     golombParameter = 10
            # make a golomb search in long list, it returns overall offset
            y = golomb_search(longList, x, golombParameter)

            if (verbose):
                print " return eden overall offset : ", y

            # if elements are equal then append
            if (x == longList[y]):
                intersectedArray.append(x)
            else:
                if (verbose):
                    print " target was ", x, " found was ", longList[y]

        return intersectedArray

    def svs_intersect(self, list_of_input_lists):

        # sort list wrt its length
        listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)

        numberOfLists = len(listsAreSortedWRTLength)

        # get shortest list
        shortestList = listsAreSortedWRTLength[0]

        intersectedArray = shortestList

        for i in range(1, numberOfLists):
            # DONE # TODO burada tekrar binary intersect'e giriyor tekrar tekra
            intersectedArray = self.binary_intersect([intersectedArray, listsAr
            # if returned array is empty no more intersection needed.
            if(len(intersectedArray)==0):
                return intersectedArray

        return intersectedArray

    def adp_intersect(self, list_of_input_lists, verbose=False):

        def getListsAsSortedWRTLength(listsOfLists):
            return sorted(listsOfLists, key=len)

        def getEliminator(lst):
```

```
            return lst.pop(0)

        intersectedArray = []

        loil = list_of_input_lists

        while (1):
            # TODO burada her seferinde listeleri sort ediyor.
            listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)

            # if first array is empty return
            if (len(listsAreSortedWRTLength[0]) == 0):
                return intersectedArray

            eliminator = getEliminator(listsAreSortedWRTLength[0])

            numberOfLists = len(listsAreSortedWRTLength)

            if (verbose):
                print eliminator

            unmatch = False
            for i in range(1, numberOfLists):

                '''
                it goes find an exact match or bigger match.
                if it finds exact match return True and restoflist
                if it finds bigger match return False and restoflist
                '''
                #bool, newList = linearSearch(listsAreSortedWRTLength[i],elimin
                bool, newList = binary_search2(listsAreSortedWRTLength[i], 0, l
                                                eliminator)

                if (bool):
                    # replace list and keep continue
                    listsAreSortedWRTLength[i] = newList
                else:
                    # unmatch case or empty list case
                    if (len(newList) == 0):
                        # list is empty
                        return intersectedArray
                    else:
                        listsAreSortedWRTLength[i] = newList
                        unmatch = True
                        break
            if (not unmatch):
                intersectedArray.append(eliminator)
            loil = listsAreSortedWRTLength
```

```

    def seq_intersect(self, list_of_input_lists):

        def getListsAsSortedWRTLength(listsOfLists):
            return sorted(listsOfLists, key=len)

        def getEliminator(lst):
            return lst.pop(0)

        intersectedArray = []
        loil = list_of_input_lists

        # initially take the shortest array as eliminator
        listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)

        if (len(listsAreSortedWRTLength[0]) == 0):
            return intersectedArray

        eliminator = getEliminator(listsAreSortedWRTLength[0])
        numberOfLists = len(listsAreSortedWRTLength)

        kingListIndex = 0
        while (1):

            #print eliminator
            unmatch = False
            for i in range(0, numberOfLists):
                ## eliminatori veren arrayi atla
                if (kingListIndex == i):
                    continue

                #bool, newList = linearSearch(listsAreSortedWRTLength[i], elimi
                bool, newList = binary_search2(listsAreSortedWRTLength[i], 0, l
                                               eliminator)

                if (bool):
                    # keep continue
                    listsAreSortedWRTLength[i] = newList
                else:
                    #  empty list case
                    if (len(newList) == 0):
                        # list is empty
                        return intersectedArray

                    # unmatch case
                    else:
                        # unmatch and list are still full.
                        # take that list as kinglist
                        listsAreSortedWRTLength[i] = newList
```

```
                        eliminator = getEliminator(listsAreSortedWRTLength[i])
                        kingListIndex = i
                        unmatch = True
                        break
                # if there is no unmatch then continue with the same list
                if (not unmatch):
                    intersectedArray.append(eliminator)
                    listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
                    if (len(listsAreSortedWRTLength[0]) == 0):
                        return intersectedArray
                    eliminator = getEliminator(listsAreSortedWRTLength[0])
                    kingListIndex = 0
                loil = listsAreSortedWRTLength



    def max_intersect(self, list_of_input_lists, verbose=False):

        def getListsAsSortedWRTLength(listsOfLists):
            return sorted(listsOfLists, key=len)

        def getEliminator(lst, popit=True):

            if (len(lst) > 0):
                if(popit):
                    return lst.pop(0)
                else:
                    return lst[0]
            else:
                return None


        lengthsorted = getListsAsSortedWRTLength(list_of_input_lists)
        intersectedArray = []

        # init values
        x = getEliminator(lengthsorted[0])
        eliminatorListIndex = 0
        # start searching from the first list initially
        startat = 1
        eliminatorArrayLength = len(lengthsorted[eliminatorListIndex])

        while (x != None):

            if(verbose):
                print "eliminator" , x , "startat list", startat,

            for i in range(startat, len(lengthsorted)):
                # print startat, x, eliminatorArrayLength
```

```
                # if any list is empty go back.
                if (len(lengthsorted[i]) == 0):
                    return intersectedArray

                # golomb search gives equal value or bigger value. if it gives
                # y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[i]

                if(len(lengthsorted[0])==0):
                    y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[
                else:

                    y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[

                if (verbose):
                    print "found y" , lengthsorted[i][y] , "in list" , lengthso

                valfound = lengthsorted[i][y]

                # remove previous value of the array,
                # arrayin yeni halinde bizim eleman artik yok
                lengthsorted[i] = lengthsorted[i][y:]

                # print len(lengthsorted[0]),len(lengthsorted[1])
                if(i==0):
                    if (valfound > x):
                        x = getEliminator(lengthsorted[0], popit=True)
                        startat = 1
                    elif (valfound < x):
                        return intersectedArray


                elif(i==1):
                    if (valfound > x):
                        x = getEliminator(lengthsorted[0], popit=False)
                        if (valfound > x):
                            dummy = getEliminator(lengthsorted[0], popit=True)
                            startat = 0
                            x = valfound
                        else:
                            x = getEliminator(lengthsorted[0], popit=True)
                            startat = 1

                        break
                    elif (valfound < x):
                        return intersectedArray
                    #eger liste 2 lik ise buraya girmemiz gerekiyor
                    elif (valfound == x):
                        if(len(lengthsorted)==2):
```

```
                            if (verbose):
                                print "found : ", x, " eliminator index ", elim
                            intersectedArray.append(x)
                            # TODO there is a problem here, will we keep get el

                            x = getEliminator(lengthsorted[0], popit=True)
                            startat = 1

                            for i in range(startat, len(lengthsorted)):
                                dmmy = getEliminator(lengthsorted[i],popit=True

                            if(verbose):
                                for i in range(len(lengthsorted)):
                                    print "arr " , i , lengthsorted[i]

                else:
                    if(valfound!=x):
                        # not found rest list
                        # get a new eliminator from 0 and start with 1
                        x = getEliminator(lengthsorted[0], popit=True)
                        startat = 1
                        break
                    else:

                        if ((i == len(lengthsorted) - 1)):

                            if (verbose):
                                print "found : ", x, " eliminator index ", elim
                            intersectedArray.append(x)
                            # TODO there is a problem here, will we keep get el

                            x = getEliminator(lengthsorted[0], popit=True)
                            startat = 1

                            for i in range(startat, len(lengthsorted)):
                                dmmy = getEliminator(lengthsorted[i],popit=True

                            if(verbose):
                                for i in range(len(lengthsorted)):
                                    print "arr " , i , lengthsorted[i]
        # eliminatorArrayLength = len(lengthsorted[eliminatorListIndex])

        return intersectedArray
```

```


def linearSearch(lst, target):
    '''
    it goes find an exact match or bigger match.
    if it finds exact match return True and restoflist
    if it finds bigger match return False and restoflist
    '''
    while (len(lst) > 0):
        if (lst[0] > target):
            return False, lst
        elif (lst[0] == target):
            lst.pop(0)
            return True, lst
        else:
            lst.pop(0)

    return False, []


def binary_search2(array, start, end, target, verbose=False):
    if start > end:
        return False,array
    middle = (start+end)/2

    if array[middle] == target:
        return True, array[middle:]
    elif array[middle] < target:
        return binary_search2(array, middle + 1, end, target)
    else:
        return binary_search2(array,start, middle-1,target)



def binary_search(array, lengthOfArray, target, verbose=False):
    if (verbose):
        print " in binary search array : ", array, " length of array ", lengthO
    if (lengthOfArray == 1):
        if (array[0] == target):
            return 0
        else:
            return lengthOfArray

    lower = 0
    upper = lengthOfArray
    while lower < upper:  # use < instead of <=
        x = lower + (upper - lower) // 2
```

```
        val = array[x]
        if target == val:
            return x
        elif target > val:
            if lower == x:  # this two are the actual lines
                return x + 1
                break  # you're looking for
            lower = x
        elif target < val:
            upper = x


def golomb_search(L, x, b, currentPosition=0, verbose=False):
    if (verbose):
        print "golomb search ==> arr : ", L, " target : ", x

    if(b==0):
        b=b+1
    curr = currentPosition
    pos = curr + b
    n = len(L) - 1

    # if x is less then first element than return directly.
    if (x < L[curr]):
        return 0

    while (pos < n and L[pos] < x):
        curr = pos
        pos = curr + b

    if (pos > n):
        pos = n

    if (verbose):
        print " element ", x, " between ", L[curr:pos + 1]

    offset = binary_search(L[curr:pos + 1], pos - curr + 1, x)
    if (verbose):
        print " calculated offset", offset

    if (curr + offset > n):
        return n
    else:
        return curr + offset
```