

# 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 exemplify 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  $0.69(n_2/n_1)$ , 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 sv<sub>s</sub> 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 sv<sub>s</sub>.

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 re-ordered 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 is it a part of the intersection's output and hence 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 sv<sub>s</sub> 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

## 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.

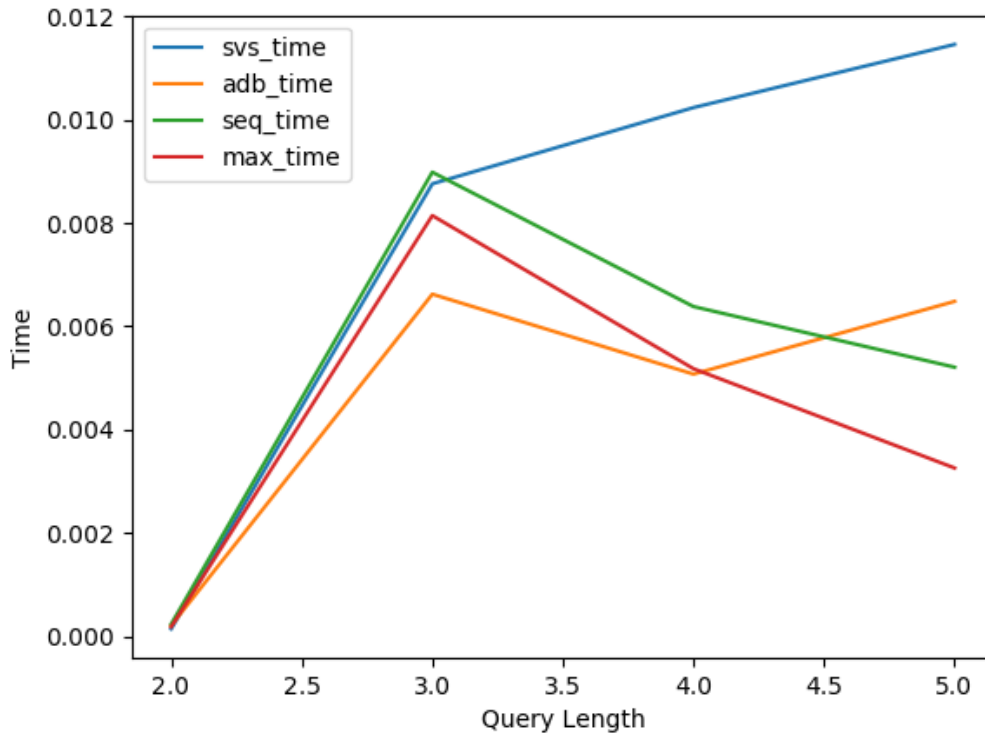


Figure 5.1: Original queries upto 5 words in a query

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.

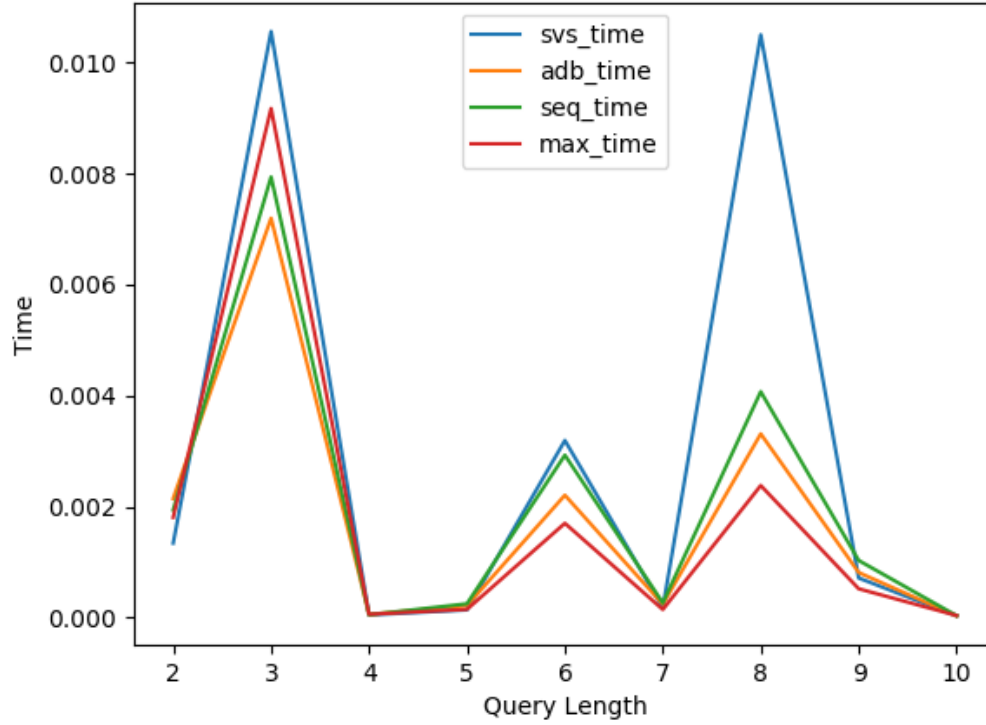


Figure 5.2: Concatenated queries upto 10 words in a query

## 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

```
1 from Searchs import *
2
3 BinaryIntersection = 0
4 SVSIntersection = 1
5 ADBIntersection = 2
6 SEQIntersection = 3
7 MAXIntersection = 4
8
9
10 class Intersections():
11     def __init__(self, type):
12         self.type = type
13
14     def intersect(self, list_of_input_lists):
15         if (self.type == BinaryIntersection):
16             return self.binary_intersect(list_of_input_lists)
17         elif (self.type == SVSIntersection):
18             return self.svs_intersect(list_of_input_lists)
19         elif (self.type == ADBIntersection):
20             return self.adp_intersect(list_of_input_lists)
21         elif (self.type == SEQIntersection):
22             return self.seq_intersect(list_of_input_lists)
23         elif (self.type == MAXIntersection):
24             return self.max_intersect(list_of_input_lists)
25
26     def binary_intersect(self, list_of_input_lists, sorrt=True, verbose=False):
27
28         assert len(list_of_input_lists) == 2
29
30         if(sorrt):
31             # sort list wrt its length
32             listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)
33         else:
34             listsAreSortedWRTLength = list_of_input_lists
35
36         intersectedArray = []
37
38         # split short and long list
39         shortList, longList = listsAreSortedWRTLength[0], \
40                               listsAreSortedWRTLength[1]
41
42
43         for ind in range(len(shortList)):
```

```

44         # calculate golomb parameter
45
46         x = shortList[ind]
47         golombParameter = int(len(longList) / (len(shortList)-ind))
48
49         # if(golombParameter > 40):
50         #     golombParameter = 10
51         # make a golomb search in long list, it returns overall offset
52         y = golomb_search(longList, x, golombParameter)
53
54         if (verbose):
55             print " return eden overall offset : ", y
56
57         # if elements are equal then append
58         if (x == longList[y]):
59             intersectedArray.append(x)
60         else:
61             if (verbose):
62                 print " target was ", x, " found was ", longList[y]
63
64         return intersectedArray
65
66     def svs_intersect(self, list_of_input_lists):
67
68         # sort list wrt its length
69         listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)
70
71         numberOfLists = len(listsAreSortedWRTLength)
72
73         # get shortest list
74         shortestList = listsAreSortedWRTLength[0]
75
76         intersectedArray = shortestList
77
78         for i in range(1, numberOfLists):
79             # DONE # TODO burada tekrar binary intersect'e giriyor tekrar tekra
80             intersectedArray = self.binary_intersect([intersectedArray, listsAr
81             # if returned array is empty no more intersection needed.
82             if(len(intersectedArray)==0):
83                 return intersectedArray
84
85         return intersectedArray
86
87     def adp_intersect(self, list_of_input_lists, verbose=False):
88
89         def getListsAsSortedWRTLength(listsOfLists):
90             return sorted(listsOfLists, key=len)
91
92         def getEliminator(lst):

```



```

93
94         return lst.pop(0)
95
96     intersectedArray = []
97
98     loil = list_of_input_lists
99
100    while (1):
101        # TODO burada her seferinde listeleri sort ediyör.
102        listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
103
104        # if first array is empty return
105        if (len(listsAreSortedWRTLength[0]) == 0):
106            return intersectedArray
107
108        eliminator = getEliminator(listsAreSortedWRTLength[0])
109
110        numberOfLists = len(listsAreSortedWRTLength)
111
112        if (verbose):
113            print eliminator
114
115        unmatched = False
116        for i in range(1, numberOfLists):
117
118            '''
119            it goes find an exact match or bigger match.
120            if it finds exact match return True and restoflist
121            if it finds bigger match return False and restoflist
122            '''
123            #bool, newList = linearSearch(listsAreSortedWRTLength[i],eliminator)
124            bool, newList = binary_search2(listsAreSortedWRTLength[i], 0, 1,
125                                           eliminator)
126
127            if (bool):
128                # replace list and keep continue
129                listsAreSortedWRTLength[i] = newList
130            else:
131                # unmatched case or empty list case
132                if (len(newList) == 0):
133                    # list is empty
134                    return intersectedArray
135                else:
136                    listsAreSortedWRTLength[i] = newList
137                    unmatched = True
138                    break
139        if (not unmatched):
140            intersectedArray.append(eliminator)
141        loil = listsAreSortedWRTLength

```

```

142
143     def seq_intersect(self, list_of_input_lists):
144
145         def getListsAsSortedWRTLength(listsOfLists):
146             return sorted(listsOfLists, key=len)
147
148         def getEliminator(lst):
149             return lst.pop(0)
150
151         intersectedArray = []
152         loil = list_of_input_lists
153
154         # initially take the shortest array as eliminator
155         listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
156
157         if (len(listsAreSortedWRTLength[0]) == 0):
158             return intersectedArray
159
160         eliminator = getEliminator(listsAreSortedWRTLength[0])
161         numberOfLists = len(listsAreSortedWRTLength)
162
163         kingListIndex = 0
164         while (1):
165
166             #print eliminator
167             unmatched = False
168             for i in range(0, numberOfLists):
169                 ## eliminatori veren arrayi atla
170                 if (kingListIndex == i):
171                     continue
172
173                 #bool, newList = linearSearch(listsAreSortedWRTLength[i], elimi
174                 bool, newList = binary_search2(listsAreSortedWRTLength[i], 0, l
175                                                 eliminator)
176
177                 if (bool):
178                     # keep continue
179                     listsAreSortedWRTLength[i] = newList
180                 else:
181                     # empty list case
182                     if (len(newList) == 0):
183                         # list is empty
184                         return intersectedArray
185
186                     # unmatched case
187                     else:
188                         # unmatched and list are still full.
189                         # take that list as kinglist
190                         listsAreSortedWRTLength[i] = newList

```

```

191         eliminator = getEliminator(listsAreSortedWRTLength[i])
192         kingListIndex = i
193         unmatched = True
194         break
195     # if there is no unmatched then continue with the same list
196     if (not unmatched):
197         intersectedArray.append(eliminator)
198         listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
199         if (len(listsAreSortedWRTLength[0]) == 0):
200             return intersectedArray
201         eliminator = getEliminator(listsAreSortedWRTLength[0])
202         kingListIndex = 0
203     loil = listsAreSortedWRTLength
204
205
206
207 def max_intersect(self, list_of_input_lists, verbose=False):
208
209     def getListsAsSortedWRTLength(listsOfLists):
210         return sorted(listsOfLists, key=len)
211
212     def getEliminator(lst, popit=True):
213
214         if (len(lst) > 0):
215             if(popit):
216                 return lst.pop(0)
217             else:
218                 return lst[0]
219         else:
220             return None
221
222
223     lengthsorted = getListsAsSortedWRTLength(list_of_input_lists)
224     intersectedArray = []
225
226     # init values
227     x = getEliminator(lengthsorted[0])
228     eliminatorListIndex = 0
229     # start searching from the first list initially
230     startat = 1
231     eliminatorArrayLength = len(lengthsorted[eliminatorListIndex])
232
233     while (x != None):
234
235         if(verbose):
236             print "eliminator" , x , "startat list", startat,
237
238         for i in range(startat, len(lengthsorted)):
239             # print startat, x, eliminatorArrayLength

```

```

240
241     # if any list is empty go back.
242     if (len(lengthsorted[i]) == 0):
243         return intersectedArray
244
245     # golomb search gives equal value or bigger value. if it gives
246     # y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[i]
247
248     if(len(lengthsorted[0])==0):
249         y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[
250     else:
251
252         y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[
253
254     if (verbose):
255         print "found y" , lengthsorted[i][y] , "in list" , lengthso
256
257     valfound = lengthsorted[i][y]
258
259     # remove previous value of the array,
260     # arrayin yeni halinde bizim eleman artık yok
261     lengthsorted[i] = lengthsorted[i][y:]
262
263     # print len(lengthsorted[0]),len(lengthsorted[1])
264     if(i==0):
265         if (valfound > x):
266             x = getEliminator(lengthsorted[0], popit=True)
267             startat = 1
268         elif (valfound < x):
269             return intersectedArray
270
271
272     elif(i==1):
273         if (valfound > x):
274             x = getEliminator(lengthsorted[0], popit=False)
275             if (valfound > x):
276                 dummy = getEliminator(lengthsorted[0], popit=True)
277                 startat = 0
278                 x = valfound
279             else:
280                 x = getEliminator(lengthsorted[0], popit=True)
281                 startat = 1
282
283             break
284         elif (valfound < x):
285             return intersectedArray
286         #eger liste 2 lik ise buraya girmemiz gerekiyor
287         elif (valfound == x):
288             if(len(lengthsorted)==2):

```

```

289         if (verbose):
290             print "found : ", x, " eliminator index ", elim
291             intersectedArray.append(x)
292             # TODO there is a problem here, will we keep get el
293
294         x = getEliminator(lengthsorted[0], popit=True)
295         startat = 1
296
297         for i in range(startat, len(lengthsorted)):
298             dmmmy = getEliminator(lengthsorted[i], popit=True)
299
300             if(verbose):
301                 for i in range(len(lengthsorted)):
302                     print "arr " , i , lengthsorted[i]
303
304         else:
305             if(valfound!=x):
306                 # not found rest list
307                 # get a new eliminator from 0 and start with 1
308                 x = getEliminator(lengthsorted[0], popit=True)
309                 startat = 1
310                 break
311             else:
312
313                 if ((i == len(lengthsorted) - 1)):
314
315                     if (verbose):
316                         print "found : ", x, " eliminator index ", elim
317                         intersectedArray.append(x)
318                         # TODO there is a problem here, will we keep get el
319
320                     x = getEliminator(lengthsorted[0], popit=True)
321                     startat = 1
322
323                     for i in range(startat, len(lengthsorted)):
324                         dmmmy = getEliminator(lengthsorted[i], popit=True)
325
326                     if(verbose):
327                         for i in range(len(lengthsorted)):
328                             print "arr " , i , lengthsorted[i]
329                 # eliminatorArrayLength = len(lengthsorted[eliminatorListIndex])
330
331         return intersectedArray

```

## 7.2 SEARCH ALGORITHMS IMPLEMENTATION

```
1
2
3 def linearSearch(lst, target):
4     '''
5     it goes find an exact match or bigger match.
6     if it finds exact match return True and restoflist
7     if it finds bigger match return False and restoflist
8     '''
9     while (len(lst) > 0):
10         if (lst[0] > target):
11             return False, lst
12         elif (lst[0] == target):
13             lst.pop(0)
14             return True, lst
15         else:
16             lst.pop(0)
17
18     return False, []
19
20
21 def binary_search2(array, start, end, target, verbose=False):
22     if start > end:
23         return False, array
24     middle = (start+end)/2
25
26     if array[middle] == target:
27         return True, array[middle:]
28     elif array[middle] < target:
29         return binary_search2(array, middle + 1, end, target)
30     else:
31         return binary_search2(array, start, middle-1, target)
32
33
34
35 def binary_search(array, lengthOfArray, target, verbose=False):
36     if (verbose):
37         print " in binary search array : ", array, " length of array ", lengthOfArray
38     if (lengthOfArray == 1):
39         if (array[0] == target):
40             return 0
41         else:
42             return lengthOfArray
43
44     lower = 0
45     upper = lengthOfArray
46     while lower < upper: # use < instead of <=
47         x = lower + (upper - lower) // 2
```

```

48         val = array[x]
49         if target == val:
50             return x
51         elif target > val:
52             if lower == x: # this two are the actual lines
53                 return x + 1
54                 break # you're looking for
55             lower = x
56         elif target < val:
57             upper = x
58
59
60 def golomb_search(L, x, b, currentPosition=0, verbose=False):
61     if (verbose):
62         print "golomb search ==> arr : ", L, " target : ", x
63
64     if(b==0):
65         b=b+1
66     curr = currentPosition
67     pos = curr + b
68     n = len(L) - 1
69
70     # if x is less then first element than return directly.
71     if (x < L[curr]):
72         return 0
73
74     while (pos < n and L[pos] < x):
75         curr = pos
76         pos = curr + b
77
78     if (pos > n):
79         pos = n
80
81     if (verbose):
82         print " element ", x, " between ", L[curr:pos + 1]
83
84     offset = binary_search(L[curr:pos + 1], pos - curr + 1, x)
85     if (verbose):
86         print " calculated offset", offset
87
88     if (curr + offset > n):
89         return n
90     else:
91         return curr + offset

```