

Efficient set intersection for inverted indexing paper report

Ozgur GUNDOGAN

January 2, 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 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 tradeoff is 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 shared by 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, with this "halving" phase having the same cost as the "doubling" phase that preceded it. 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.

The search algorithm can also be based on Golomb searching, in a mechanism described by Hwang and Lin [1972]. Search proceeds in a manner somewhat similar to exponential search, but with a fixed forwards step of b items used at each iteration. Once overshoot has been achieved, a binary search takes place over the (at most) 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 mentality. 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 svcs approach is to combine all of the sets using a single concerted sweep through them all. The resultant holistic algorithms offer the possibility of being adaptive to the particular data arrangement present, and can potentially outperform the svcs approaches. 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 svcs.

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 $|q|$ 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), 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 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 svcs methods, because all of the sets are processed concurrently. To ameliorate this risk, we 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, while it is true that in the absence of other information, the best eliminator will arise in the smallest set, the likelihood of another set becoming significantly smaller than S_1 during processing is small. The second intuition is that, having discovered a bigger than anticipated jump in one of the sets, that value should naturally be tested against the first set, to see if additional items can be discarded.

4 SOURCE CODE

4.1 INTERSECTION ALGORITHMS IMPLEMENTATION

```
1
2 from Searchs import *
3
4
5 BinaryIntersection = 0
6 SVSIntersection = 1
7 ADBIntersection = 2
8 SEQIntersection = 3
9 MAXIntersection = 4
10
11
12 class Intersections():
13
14     def __init__(self,type):
15         self.type = type
16
17     def intersect(self,list_of_input_lists):
18         if(self.type==BinaryIntersection):
19             return self.binary_intersect(list_of_input_lists)
20         elif(self.type==SVSIntersection):
21             return self.svs_intersect(list_of_input_lists)
22         elif (self.type == ADBIntersection):
23             return self.adp_intersect(list_of_input_lists)
24         elif (self.type == SEQIntersection):
25             return self.seq_intersect(list_of_input_lists)
26         elif (self.type == MAXIntersection):
27             return self.max_intersect(list_of_input_lists)
28
29     def binary_intersect(self,list_of_input_lists, verbose = False):
30
31         assert len(list_of_input_lists)==2
32
33         listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)
34         intersectedArray = []
35
36         shortList,longList = listsAreSortedWRTLength[0] , listsAreSortedWRTLength[1]
37         for x in shortList:
38             golombParameter = int(len(longList)/len(shortList))
39             y = golomb_search(longList,x,golombParameter)
40             if(verbose):
41                 print " return eden overall offset : " , y
42             if (x==longList[y]):
43                 intersectedArray.append(x)
44             else:
45                 if (verbose):
```

```

46         print " target was " , x , " found was " , longList[y]
47
48
49     return intersectedArray
50
51
52     def svs_intersect(self,list_of_input_lists):
53
54         listsAreSortedWRTLength = sorted(list_of_input_lists , key=len)
55
56         numberOfLists = len(listsAreSortedWRTLength)
57
58         shortestList = listsAreSortedWRTLength[0]
59         intersectedArray = shortestList
60
61         for i in range(1, numberOfLists):
62
63             intersectedArray = self.binary_intersect([intersectedArray,listsAre
64
65
66         return intersectedArray
67
68     def adp_intersect(self,list_of_input_lists,verbose=False):
69
70         def getListsAsSortedWRTLength(listsOfLists):
71             return sorted(listsOfLists, key=len)
72
73         def getEliminator(lst):
74
75             return lst.pop(0)
76
77         intersectedArray = []
78         loil = list_of_input_lists
79
80         while(1):
81             listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
82             if(len(listsAreSortedWRTLength[0]) == 0):
83                 return intersectedArray
84             eliminator = getEliminator(listsAreSortedWRTLength[0])
85
86             numberOfLists = len(listsAreSortedWRTLength)
87             if(verbose):
88                 print eliminator
89
90             unmatched = False
91             for i in range(1,numberOfLists):
92                 bool, newList =linearSearch(listsAreSortedWRTLength[i],eliminat
93
94                 if(bool):

```

```

95         #keep continue
96         listsAreSortedWRTLength[i] = newList
97     else:
98         # unmatched case or empty list case
99         if(len(newList)==0):
100             # list is empty
101             return intersectedArray
102         else:
103             listsAreSortedWRTLength[i] = newList
104             unmatched = True
105             break
106     if(not unmatched):
107         intersectedArray.append(eliminator)
108     loil = listsAreSortedWRTLength
109
110 def seq_intersect(self, list_of_input_lists):
111
112     def getListsAsSortedWRTLength(listsOfLists):
113         return sorted(listsOfLists, key=len)
114
115     def getEliminator(lst):
116         return lst.pop(0)
117
118     intersectedArray = []
119     loil = list_of_input_lists
120
121     # initially take the shortest array as eliminator
122     listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
123     if (len(listsAreSortedWRTLength[0]) == 0):
124         return intersectedArray
125     eliminator = getEliminator(listsAreSortedWRTLength[0])
126     numberOfLists = len(listsAreSortedWRTLength)
127
128     kingListIndex = 0
129     while (1):
130
131         print eliminator
132         unmatched = False
133         for i in range(0, numberOfLists):
134             ## eliminatori veren arrayi atla
135             if(kingListIndex == i ):
136                 continue
137
138
139             bool, newList = linearSearch(listsAreSortedWRTLength[i], elimin
140
141             if (bool):
142                 # keep continue
143                 listsAreSortedWRTLength[i] = newList

```

```

144         else:
145             # unmatched case or empty list case
146             if (len(newList) == 0):
147                 # list is empty
148                 return intersectedArray
149             else:
150                 listsAreSortedWRTLength[i] = newList
151                 eliminator = getEliminator(listsAreSortedWRTLength[i])
152                 kingListIndex = i
153                 unmatched = True
154                 break
155     # if there is no unmatched then continue with the same list
156     if (not unmatched):
157         intersectedArray.append(eliminator)
158         listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
159         if (len(listsAreSortedWRTLength[0]) == 0):
160             return intersectedArray
161         eliminator = getEliminator(listsAreSortedWRTLength[0])
162         kingListIndex = 0
163     loil = listsAreSortedWRTLength
164
165
166     def max_intersect(self, list_of_input_lists):
167
168         def getEliminator(lst):
169
170             if (len(lst) > 0):
171                 return lst.pop(0)
172             else:
173                 return None
174
175         lengthsorted = sorted(list_of_input_lists, key=len)
176         intersectedArray = []
177
178
179         eliminatorArrayLength = len(lengthsorted[0])
180         x = getEliminator(lengthsorted[0])
181         startat = 1
182
183         while(x):
184             for i in range(startat, len(lengthsorted)):
185                 print startat, x, eliminatorArrayLength
186
187                 if (len(lengthsorted[i]) == 0):
188                     return intersectedArray
189                 y = golomb_search(lengthsorted[i], x, int(len(lengthsorted[i])/el
190                 if (lengthsorted[i][y] > x):
191                     x = getEliminator(lengthsorted[0])
192                 if (lengthsorted[i][y] > x):

```

```
193         startat=0
194         x = lengthsorted[i][y]
195     else:
196         startat=1
197
198         break
199     elif(i == len(lengthsorted)-1):
200         intersectedArray.append(x)
201         x = getEliminator(lengthsorted[0])
202         startat=1
203     return intersectedArray
```


4.2 SEARCH ALGORITHMS IMPLEMENTATION

```
1
2 def linearSearch(lst,target):
3     while(len(lst)>0):
4         if(lst[0]>target):
5             return False,lst
6         elif(lst[0] == target):
7             lst.pop(0)
8             return True, lst
9         else:
10            lst.pop(0)
11
12    return False,[]
13
14
15 def binary_search(array, lengthOfArray, target, verbose = False):
16     if (verbose):
17         print " in binary search array : ", array , " length of array ", length
18     if(lengthOfArray ==1):
19         if(array[0] == target):
20             return 0
21         else:
22             return lengthOfArray
23
24     lower = 0
25     upper = lengthOfArray
26     while lower < upper:      # use < instead of <=
27         x = lower + (upper - lower) // 2
28         val = array[x]
29         if target == val:
30             return x
31         elif target > val:
32             if lower == x:      # this two are the actual lines
33                 return x+1
34                 break          # you're looking for
35             lower = x
36         elif target < val:
37             upper = x
38
39
40 def golomb_search(L,x,b,currentPosition=0,verbose=False):
41     if (verbose):
42         print "golomb search ==> arr : " , L ," target : " , x
43     curr = currentPosition
44     pos = curr + b
45     n = len(L)-1
46
47     # if x is less then first element than return directly.
```

```

48     if(x<L[curr]):
49         return 0
50
51     while(pos < n and L[pos]<x):
52         curr = pos
53         pos = curr + b
54
55     if(pos > n):
56         pos = n
57
58     if (verbose):
59         print " element " , x , " between " , L[curr:pos+1]
60
61     offset = binary_search(L[curr:pos+1],pos-curr + 1 ,x)
62     if (verbose):
63         print " calculated offset" , offset
64
65     if(curr+offset > n):
66         return n
67     else:
68         return curr+offset

```