MIDDLE EAST TECHNICAL UNIVERSITY, CENG 776

Efficient set intersection for inverted indexing paper report

Ozgur GUNDOGAN

January 1, 2018

1 INTRODUCTION

A conjunctive query q is equivalent to a |q|-way intersection over 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. As is the case with many computing applications, there is tension between the way in which the data is represented, and the ways in which it is to be manipulated. Our purpose in this paper is to explore these tradeoffs, by investigating intersection techniques that make use of both uncompressed "integer" representations.We also propose a simple hybrid method that provides both compact storage, and also faster intersection computations for conjunctive querying than is possible even with uncompressed representations.

2 ALGORITHMS FOR EFFICIENT F-SEARCH

Binary search over n2 elements requires 1+log n2 comparisons, and if set T is stored as a sorted array of explicit values (SAEV format), then binary search can be used to underpin the F-SEARCH operations required in Algorithm 1. In particular, binary search is the optimal approach when |S| = 1.

There are also other searching methods that can be applied to SAEV representations, including linear search, interpolation search, Fibonacci search, exponential search (also referred to as galloping search by some authors), and Golomb search [Hwang and Lin 1972]. 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; and as is described shortly, exponential search requires O(log d) time [Bentley and Yao 1976].

In situations when 1 âĽł n1 âĽł n2, use of exponential search in the F-SEARCH implementation is of 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 âĂIJhalvingâĂİ phase having the same cost as the âĂIJdoublingâĂİ phase that preceded it. In this approach each F-SEARCH call requires 1 + 2âŇŁlog dâŇŃ comparisons, where d is the difference between the rank of the fingerâĂŹs previous position and the new rank of the finger pointer. Over n1 calls for which Pn1 i=1 di âĽd' n2, the convex nature of the log function means that at most O(n1 + n1 log(n2/n1)) comparisons are required. Note that this approach has the same worst case asymptotic cost as using binary search when n1 is O(1), and has the same worst case asymptotic cost as linear search when n2/n1 is O(1).

The F-SEARCH algorithm can also be based on Golomb searching, in a mechanism described by Hwang and Lin [1972]. Algorithm 2 shows an implementation of this technique. Search proceeds in a manner somewhat similar to exponential F-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 n2 for the elements of a set of size n1, the correct value for the step b is 0.69(n2/n1), with a total search cost that is again proportional to O(n1 + n1 log(n2/n1)) [Gallager and van Voorhis 1975].

3 INTERSECTION METHODS

3.1 BINARY INTERSECTION OF ORDERED SETS

Algorithm 1 describes a more complex but also more efficient intersection algorithm, in which each element of the smaller set, S, is tested against the larger set, T , and retained if it is present [Hwang and Lin 1972]. 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. An auxiliary operation, FIRST, is used to establish an initial state in S; and T is implicitly assumed to have also been initialized, so that the first F-SEARCH starts from its least item.

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. Algorithm 3 shows this small versus small (svs) approach. The smallest set is identified, and then that set is intersected with each of the others, in increasing order of size. The candidate set is never larger than S1 was initially, so the worst-case cost of this approach using a SAEV data representation using an F-SEARCH that takes O(log d) time to process a jump of length d is given by X |q| i=2 n1 log ni n1 âĽď n1(|q| âĹŠ 1)log n|q| n1, where it is assumed that the sets are ordered by size, with n1 âĽď n2 âĽď Âů ÂůÂů âĽď n|q| 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 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 svs approaches. Still working with the SAEV representation, 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. Other adaptive approaches have been proposed which primarily differ in the way that the eliminators are selected at each iteration.

In 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

Their sequential algorithm, denoted here as 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 âĂŞ and hence is part of the intersectionâĂŹs 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 ameliorate this risk, we propose a further alternative, described by Algorithm 4. 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 S2 if the eliminator is again taken from S1, otherwise processing begins in S1. 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 S1 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

```

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, verbose = False):

        assert len(list_of_input_lists)==2

        listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)
        intersectedArray = []

        shortList,longList = listsAreSortedWRTLength[0] , listsAreSortedWRTLeng
        for x in shortList:
            golombParameter = int(len(longList)/len(shortList))
            y = golomb_search(longList,x,golombParameter)
            if(verbose):
                print " return eden overall offset : " , y
            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):

        listsAreSortedWRTLength = sorted(list_of_input_lists, key=len)

        numberOfLists = len(listsAreSortedWRTLength)

        shortestList = listsAreSortedWRTLength[0]
        intersectedArray = shortestList

        for i in range(1, numberOfLists):

            intersectedArray = self.binary_intersect([intersectedArray,listsAre


        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):
            listsAreSortedWRTLength = getListsAsSortedWRTLength(loil)
            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):
                bool, newList =linearSearch(listsAreSortedWRTLength[i],eliminat

                if(bool):
```

```
                        #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], elimin

                if (bool):
                    # 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
                        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):

        def getEliminator(lst):

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

        lengthsorted = sorted(list_of_input_lists, key=len)
        intersectedArray = []


        eliminatorArrayLength = len(lengthsorted[0])
        x = getEliminator(lengthsorted[0])
        startat = 1

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

                if(len(lengthsorted[i])==0):
                    return intersectedArray
                y = golomb_search(lengthsorted[i],x,int(len(lengthsorted[i])/el
                if(lengthsorted[i][y]>x):
                    x = getEliminator(lengthsorted[0])
                    if(lengthsorted[i][y]>x):
```

```
                        startat=0
                        x = lengthsorted[i][y]
                    else:
                        startat=1

                    break
                elif(i == len(lengthsorted)-1):
                    intersectedArray.append(x)
                    x = getEliminator(lengthsorted[0])
                    startat=1
        return intersectedArray
```

4.2 Search Algorithms Implementation

```

def linearSearch(lst,target):
    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_search(array, lengthOfArray, target, verbose = False):
    if (verbose):
        print " in binary search array : ", array , " length of array ", length
    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
    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
```