Computing the Top-k Maximal Answers in a Join of Ranked Lists[*]

Mirit Shalem
Faculty of Computer Science, Technion
Haifa, Israel
mirit2s@cs.technion.ac.il

Yaron Kanza
Faculty of Computer Science, Technion
Haifa, Israel
kanza@cs.technion.ac.il

ABSTRACT

Complex search tasks that utilize information from several data sources, are answered by integrating the results of distinct basic search queries. In such integration, each basic query returns a ranked list of items, and the main task is to compute the join of these lists, returning the top-k *combinations*. Computing the top-k join of ranked lists has been studied extensively for the case where the answer comprises merely complete combinations. However, a join is a lossy operation, and over heterogeneous data sources some highly-ranked items, from the results of the basic queries, may not appear in any combination. Yet, such items and the partial combinations in which they appear may still be relevant answers and should not be discarded categorically.

In this paper we consider a join where combinations are padded by nulls for missing items. A combination is *maximal* if it cannot be extended by replacing a null by an item. We present algorithms for computing the top-k maximal combinations and provide an experimental evaluation.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Performance

Keywords

Data integration, search, top-k, join, maximal answers

1. INTRODUCTION

Search is a fundamental service provided by numerous applications. Usually, a *basic* search application returns a ranked list of relevant items by posing a single query over a single data source. Yet, some complex search tasks require combining the answers of several basic search queries, by joining the ranked lists [1]. The result of a join is a set of *combinations*, where each combination is a set of related items—comprising a single item from each list. Combinations are ranked according to the ranking scores of the items they comprise.

When integrating information from different sources, the algorithms should be able to cope with missing data, and should avoid discarding data from one source simply because it does not have a matching data in another source.

In *incomplete combinations*, nulls fill in for missing items. However, in order to provide as much relevant information as possible, the result comprises only combinations that are *maximal*, in the sense that none of these combinations is a proper subset of any other combination. Requiring maximality prevents duplications such as having in the result several subsets of the same combination.

Computing the join of ranked lists, for the case where the result contains only complete combinations, is a well-studied subject. Ilyas et al. proposed a method that uses pipelining and binary join operators, for computing equijoin of ranked lists [2]. This approach has been further investigated by Schnaitter and Polyzotis [3]. In this paper, we show how their algorithm can be modified to return maximal rather than complete combinations. We refer to the modified algorithms by the name HNL. We also introduce two additional join algorithms that return maximal answers. One algorithm, namely VNL, is more efficient than HNL for the case where associations between items in the lists are infrequent. The second algorithm is a hybrid between VNL and HNL and it outperforms both in almost all cases.

In this paper, we present our join algorithms and the optimizations that were used for an efficient implementation of these algorithms. Experimental evaluation illustrates the benefits of each algorithm.

2. FRAMEWORK

Combined Search Query. A *combined search query* Q comprises several basic search queries $Q = (q_1, q_2, \ldots, q_t)$. We refer to $q_1, \ldots, q_t$ as the *subqueries* of Q.

A preliminary step for evaluating a combined search query Q is to evaluate all its subqueries. The result of this step is the *pre-answer* of Q, $PreAns(Q) = (L_1, \ldots, L_t)$, where L_j is the answer to q_j. L_j is a list of items $(o_1^j, o_2^j, \ldots, o_{m_j}^j)$, sorted in descending order by their *relevance score*.

Complete Combinations. Given a combined query Q,

[*]This work was partially supported by GIF grant 2165-1738.6/2007 and by MOST grant 3/6472.

items of different lists of the pre-answer are joined to form combinations. The join operation is based on conditions which determine whether the items should be joined. We refer to a pair of items that are joined as associated. The predicate $JCon(o_i, o_j)$ is true when o_i and o_j are associated, and false otherwise.

A *complete combination* is a t-tuple of items,

$$C = (o^1_{i_1}, o^2_{i_2}, \ldots, o^t_{i_t}) \in L_1 \times L_2 \cdots \times L_t$$

such that each item is an element of the corresponding list of the pre-answer, and every pair of items in C is associated. We denote by $Join(Q)$ the set of all the combinations that are produced by joining the lists of the pre-answer of Q.

The *score* of a combination C, denoted $score(C)$, is the result of a monotonic function over the item scores.

Maximal Combinations. We represent missing values using nulls, denoted by the $\perp$ symbol. A *partial combination* is a t-tuple of elements, where each element is either an item of the corresponding list or a null value, and every pair of non-null items is associated. When computing the score of a partial combination, a null value is considered as an item whose score is zero.

A partial combination $C_1 = (o_{i_1}, o_{i_2}, \ldots, o_{i_t})$ *subsumes* a partial combination $C_2 = (o_{j_1}, o_{j_2}, \ldots, o_{j_t})$, denoted $C_1 \succ C_2$, if for every $1 \le k \le t$, either $o_{j_k} = \perp$ or $o_{j_k} = o_{i_k}$. That is, every non-null item in C_2 is equal to the corresponding item in C_1.

A combination is *maximal* if it is not a proper subset of any other combination. Formally, a combination C is maximal if there is no combination $C' \in (L_1 \cup \{\perp\}) \times \cdots \times (L_t \cup \{\perp\})$ such that $C' \neq C$ and $C' \succ C$.

We denote by $MaxJoin(Q)$ the set of all maximal combinations that are produced by joining the lists of the pre-answer of Q. The *top-k maximal answers* of Q are the k combinations with the highest scores in $MaxJoin(Q)$.

3. ALGORITHMS

In this section we present algorithms for computing the top-k maximal answers for a given combined search query Q. Throughout this section, we assume that the input consists of the lists $L_1, \ldots, L_t$ of $PreAns(Q)$. The initial step is to modify these lists by adding a null value with score zero at the end of each list. We denote these extended lists by $L^+_1, \ldots, L^+_t$, i.e., $L^+_j = L_j \cup \{\perp\}$, for $j = 1 \ldots t$. The join condition is such that a null value, in any list, is associated with any item and with any null of any other list.

The algorithms we present manage a heap of size (at most) k that stores combinations. We denote by $MinScore(H)$ a function that returns the minimal score of a combination in H if the heap is full, or zero in the case that H currently contains less than k combinations.

We denote by $H.add(C)$ the two-step addition operation that (1) adds a combination C to H, if there is no other combination in H that subsumes C; and (2) if after the addition there are $k+1$ elements in H, removes from H the combination with the lowest score.

3.1 The Horizontal Nested Loop Algorithm

The first algorithm we consider is essentially a modification of the algorithm of Ilyas et al. [2], where instead of applying it over $L_1, \ldots, L_t$, we apply it over the lists $L^+_1, \ldots, L^+_t$. Given lists $L_1, \ldots, L_t$ and a value k, the algorithm iterates over the lists, horizontally, and in each iteration adds the discovered combinations to the heap H. We denote this algorithm by HNL.

Initially, the algorithm visits the first item of each list, and in the i-th iteration it visits the i-th item of each list. When visiting an item o^j_i of list j, the algorithm joins the singleton $\{o^j_i\}$ with all the items of the other lists that were visited before o^j_i. That is, the algorithm computes the join of the lists $T_{i,1}, \ldots, T_{i,j-1}, \{o^j_i\}, T_{i-1,j+1}, \ldots, T_{i-1,t}$, where $T_{i,j}$ are the top-i items of L^+_j, and it adds the produced combinations to H.

For computing the join in each iteration, the algorithm employs the vertical nested loop algorithm (VNL) that is presented in the following section (Section 3.2). As an optimization, it takes the singleton list as the outermost list, in the nested loop.

In order to apply early termination, when for some visited item o^j_i it holds that $score(o^1_1, \ldots, o^{j-1}_1, o^j_i, o^{j+1}_1, \ldots, o^t_1) \le MinScore(H)$, the algorithm stops visiting items of list j. That is, let τ^j_i be the tuple that contains o^j_i and the first item o^k_1 of list L^+_k, for all $k = 1 \ldots j-1, j+1, \ldots t$. If the score of τ^j_i is lower than the score of the lowest combination in the heap H, then there is no additional possible combination that can be added to H, among the combinations that contain an item lower than o^j_i in L^+_j. Thus, the algorithm stops visiting items below o^j_i in L^+_j. When in all the lists the algorithm either reaches this stopping condition or reaches the end of the list, the computation terminates.

Note that in HNL, all the complete combinations are constructed prior to the construction of partial combinations.

3.2 The Vertical Nested Loop Algorithm

The *Vertical Nested-Loop Algorithm* (Algorithm 1), VNL for short, is an optimized version of an algorithm that iterates over the lists in a vertical fashion. Essentially, VNL iterates through the lists by nested loops and maintains the top combinations in a heap H. Whenever it finds a combination C, it inserts C to H by applying $H.add(C)$, (i.e., C is being added to H only if there is no combination in H that subsumes C, and the score of C is higher than $MinScore(H)$). In order to increase the efficiency, VNL breaks the loops in cases where continuing the iteration would not be able to produce relevant combinations, i.e., combinations that will be inserted to the top-k heap. Consequently, VNL checks less combinations than a naive nested loop algorithm.

In HNL, a partial combination can be inserted into H only after all the relevant complete combinations were added to H. This is not the case in VNL. A partial combination may be added to the heap H prior to the insertion of some complete combination. Yet, in each step of the algorithm all the combinations in H are maximal combinations. This is because when a combination C is added to H, there cannot be in H a combination that subsumes C.

4. THE HYBRID APPROACH

Consider the two algorithms VNL and HNL that we have discussed so far. An analysis can show that VNL is more efficient than HNL when the percentage of associations is low, i.e., when there are relatively few associated pairs. HNL is more efficient than VNL when the percentage of associations is high. The reason is that VNL may read, in such case, much more items than HNL in its inner loops.

Algorithm 1 Vertical Nested Loops (VNL)

```
Input: $L_1, \cdots, L_t; k$
Output: Top-k maximal combinations
 1: add an item $\perp$ with score 0 to the end of each list, creating $L_1^+, \cdots, L_t^+$
 2: $H := \emptyset$                                   ▷ a heap of size $k$
 3: for $a_1$ in $L_1^+$ do
 4:     if $score(a_1, o_1^2, \ldots, o_1^t) < MinScore(H)$ then
 5:         break
 6:     end if
 7:     ...
 8:     for $a_i$ in $L_i^+$ do
 9:         let $s = score(a_1, \ldots, a_i, o_1^{i+1}, \ldots, o_1^t)$
10:         if $s < MinScore(H)$ then
11:             break
12:         end if
13:         if exists $1 \leq j \leq i-1$ s.t. not $JCon(a_j, a_i)$ then
14:             continue
15:         end if
16:         ...
17:         for $a_t$ in $L_t^+$ do
18:             if $JCon(a_t, a_j) = true$ for $1 \leq j \leq t-1$ then
19:                 $newComb := (a_1, \ldots, a_t)$
20:                 $H.add(newComb)$
21:             end if
22:         end for
23:         ...
24:     end for
25:     ...
26: end for
```

The hybrid approach tries to combine the benefits of both algorithms for producing an algorithm that is at least as efficient as HNL in the case where there are many associations between items, and at least as efficient as VNL when there are a few associations between items.

Algorithm VHNL is a hybrid, between VNL and HNL, that works as follows. It computes a value h, iterates over the h first elements of each list using VNL and then continues by iterating over the lists horizontally using HNL, starting from the $(h+1)$-st row. Note that when $h = 0$, the hybrid algorithm VHNL immediately applies HNL, while for $h \geq m$, where m is the number of items in the longest list, VHNL calls only to VNL, without switching to HNL.

Our goal is to choose h such that VHNL will outperform both VNL and HNL. We do that as follows. Given the lists $L_1, \ldots, L_t$, we denote by P_{ij} the probability that a pair (o_i, o_j), of uniformly selected items $o_i \in L_i$ and $o_j \in L_j$, are associated. That is, if A_{ij} is the set of associated pairs of items in $L_i \times L_j$, then

$$P_{ij} = \frac{|A_{ij}|}{|L_i \times L_j|}.$$

The probability that some randomly chosen tuple among the tuples of $L_1 \times L_2 \times \cdots \times L_t$ is a combination is

$$P_C = \Pi_{1 \leq i < j \leq t}(P_{ij}).$$

Thus, h items from each list are expected to yield $h^t \cdot P_C$ combinations. Since k combinations fill the top-k heap, we

Algorithm 2 VHNL.

```
Input: $L_1, \ldots, L_t; k$
Output: Top-k maximal combinations
1: $H := \emptyset$                                   ▷ a heap of size $k$
2: Compute $h$ according to Equation 1
3: For $i = 1, \ldots, t$ let $V_i$ be the top $h$ items in $L_i$
4: Compute VNL($V_1$, ..., $V_t$, $k$, $H$)
5: Compute HNL($L_1$, ..., $L_t$, $k$, $H$) starting from row $h+1$
6: return $H$
```

take $k = h^t \cdot P_C$, and we compute h as follows:

$$h = \sqrt[t]{\frac{k}{P_C}} \qquad (1)$$

Algorithm 2 presents VHNL. It computes h using Equation 1, and applies VNL and HNL accordingly. The P_{ij}-s probabilities for the computation of h are either known a priori or being estimated using sampling.

5. EXPERIMENTAL EVALUATION

We conducted a variety of experiments to evaluate the performance of the presented algorithms. In what follows, we first describe the experimental methodology and then present the results of the evaluation.

5.1 Methodology

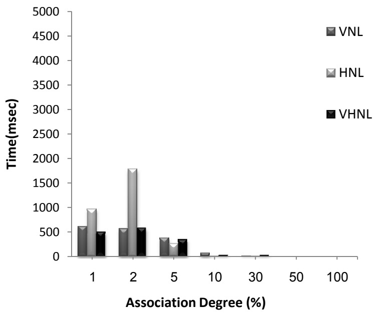
The experiments were conducted on a 2.0GHz Pentium 4 machine with 2GB RAM, running the Windows XP operating system. The algorithms were implemented in Java. In order to be able to vary parameters on demand we utilized synthetic data sets. We generated lists of synthetic items, controlling various parameters (e.g., number of lists, their lengths, association degree—the probability that two items are joined). We also tested our techniques on real geospatial data set. The data is part of a digital map of the city Tel-Aviv. The data set consists of 400 geographical objects in four lists (cinemas, hotels, pharmacies and synagogues). Each object has a relevance score, latitude and longitude. We joined two items if the Euclidean distance between them is at most 200 meters.

The parameters m (list length), t (number of lists), k (number of desired outputs), and p (association degree) are specified for each experiment. The default values are $m = 1000$, $t = 4$, $k = 100$, and $p = 2\%$. The score of a combination, in both the synthetic and real data sets, is the sum of scores of the joined items.

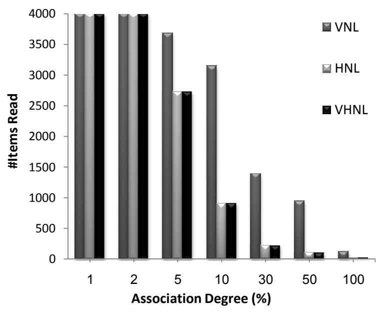
Evaluation metrics. To evaluate the performance of the algorithms, we measured, in addition to the running times (which are measured in milliseconds), the number of items scanned (#ItemsReads), and the number of pairwise association checks (#BinJoins). The first measure is the access cost of the algorithm, while the second corresponds to the join computational cost.

5.2 Results

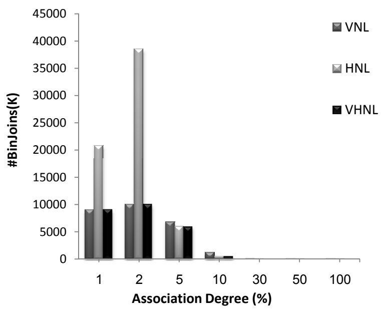
In the following experiments, the scores in every list are distributed uniformly in the range [0,100]. The first experiment studies the effect of the association degree on the performance of the algorithms over synthetic data (with default parameters). The results are presented in Figure 1. When



(a) Running time

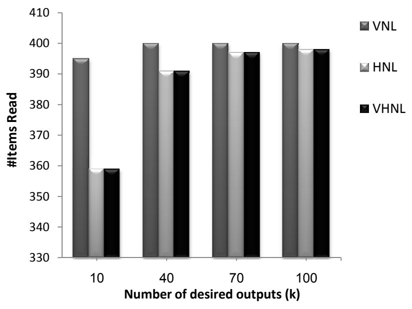


(b) Access cost

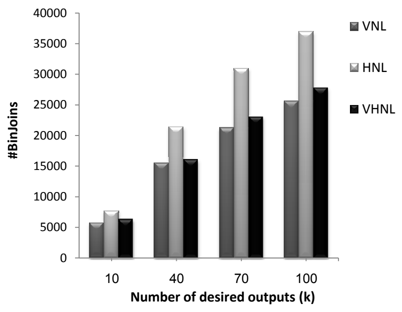


(c) Join cost

Figure 1: Effect of association degree ($m = 1000$, $t = 4$, $k = 100$)



(a) Access cost



(b) Join cost

Figure 2: Performance as a function of k over real data

the association degree is low (up to 2%), all algorithms read the entire input, which is expected as there are relatively few combinations, and the top-k ones are not necessarily located high on the lists. However, there is a significant difference in the join cost of the algorithms. VNL is more efficient than HNL, both in running time and in join cost. Note that for low association degrees, even when there are dozens of complete combinations, the top-10 answer (and obviously for larger values of k) contains partial combinations. As the association degree increases, and the top-k results are located higher in the lists, the access cost of all algorithms decreases. However, in this case the trend is reversed as HNL outperforms VNL. It reads less items and performs less join checks than VNL. VHNL is almost optimal in all degrees of association. This emphasizes the importance of estimating the depth in the lists over which VNL should be executed.

The second experiment evaluates the performance of the algorithms as we vary the desired number of outputs (i.e., k). Figure 2 shows the results over real data as a function of k. All algorithms have about the same running time (less than 16 milliseconds). VNL has the worst access cost, and HNL has the worst join cost. As expected, all costs are increased when k is increased. VHNL, again, is the most efficient when considering both access cost and join cost.

Figure 3 shows the join cost over synthetic data with low association degree, as a function of k. In this case, the algorithms read the entire input. Hence, only the join cost varies. In addition, VHNL performs the same as VNL (because h is high) and better than HNL. When k increases, the join cost of VHNL and VNL only slightly increases, because such increase only requires maintaining a larger priority queue. To conclude, VHNL is the best algorithm in almost all cases, for all measures.

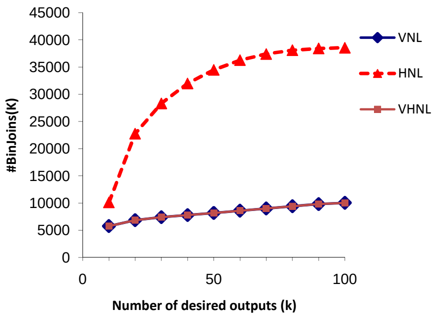


Figure 3: Join cost as a function of k ($m = 1000$, $t = 4$, $p = 2\%$)

6. REFERENCES

[1] S. Ceri and M. Barambilla. *Search Computing: Challenges and Directions*. Springer, 2010.

[2] F. Ilyas, G. Aref, and K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[3] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, pages 43–52, 2008.