Efficient Aggregation of Ranked Inputs*

Nikos Mamoulis, Kit Hung Cheng, Man Lung Yiu, and David W. Cheung
Department of Computer Science
University of Hong Kong
Pokfulam Road
Hong Kong
{nikos,khcheng,mlyiu2,dcheung}@cs.hku.hk

Abstract

A top-k query combines different rankings of the same set of objects and returns the k objects with the highest combined score according to an aggregate function. We bring to light some key observations, which impose two phases that any top-k algorithm, based on sorted accesses, should go through. Based on them, we propose a new algorithm, which is designed to minimize the number of object accesses, the computational cost, and the memory requirements of top-k search. Adaptations of our algorithm for search variants (exact scores, on-line and incremental search, top-k joins, other aggregate functions, etc.) are also provided. Extensive experiments with synthetic and real data show that, compared to previous techniques, our method accesses fewer objects, while being orders of magnitude faster.

1 Introduction

Several applications combine ordered scores of the same set of objects from different (distributed) sources and return the objects in decreasing order of their combined scores, according to an aggregate function. Assume for example that we wish to retrieve the restaurants in a city in decreasing order of their aggregate scores with respect to how cheap they are, their quality, and their closeness to our hotel. If three separate services can incrementally provide ranked lists of the restaurants based on their scores in each of the query components, the problem is to identify the k restaurants with the best combined (e.g., average) score.

This problem, known as the top-k query, has received considerable attention. Fagin's early algorithm [4], later optimized in [13, 7, 6], assumes that the score of an object x can be accessed from each source S_i both *sequentially* (i.e., after all objects with higher ranking than x have been seen there), or *randomly* by explicitly querying S_i about x. On the other hand, in this paper, we focus on top-k queries in the case where the atomic scores in each source can be accessed only in sorted order; i.e., it is not possible to know the score of an object in source S_i, before all objects better than x in S_i have been seen there. This case has received increasing interest [8, 12, 9, 10] for several reasons. First, in many applications, random accesses to scores are impossible [6]. For instance, a typical web search engine does not explicitly return the similarity between a query and a particular document in its database (it only ranks similar to the query documents). Second, even when random accesses are allowed, they are usually considerably more expensive that sorted accesses. Third, we may want to merge (possibly unbounded) streams of ranked inputs [10], produced incrementally and/or on-demand, where individual scores of random objects are not available at anytime.

Fagin et al. [6] proposed a top-k algorithm that performs "no random accesses" (NRA) and proved that it is asymptotically no worse (in terms of accesses) than any top-k method based on sorted accesses only. Nevertheless, as shown in [8, 12, 9, 10], NRA algorithms can have significant performance differences in terms of (i) accesses, (ii) computational cost, and (iii) memory requirements. The number of accesses is a significant cost factor, especially for middleware applications which charge by the amount of information transferred from the various (distributed) sources. The computational cost is critical for real-time applications, whereas memory is an issue for NRA algorithms, which, as opposed to random-access based methods (e.g., [13]), have large buffer requirements [12, 9].

The first contribution of this paper is the identification of some key observations, which have been overlooked by past research and apply on the whole family of "no random accesses" (NRA) algorithms. These observations impose two phases that any NRA algorithm should go through; a *growing* phase, during which the set of top-k candidates grows and no pruning can be performed and a *shrinking* phase,

*Work supported by grant HKU 7380/02E from Hong Kong RGC.

during which the set of candidates shrinks until the top-k result is finalized. Our second contribution is a careful implementation of a top-k algorithm, which is based on these observations and employs appropriate data structures to minimize the accesses, computational cost, and memory requirements of top-k search. Our algorithm can be implemented as a standalone rank aggregation tool or as a multi-way merge join operator for dynamically produced ranked inputs. Extensive experiments with synthetic and real data show that, compared to previous techniques, our method accesses (sometimes significantly) fewer objects, while being orders of magnitude faster.

The rest of the paper is organized as follows. In Section 2 we review related work on top-k query processing. Section 3 motivates this research and identifies some key properties on the behavior of NRA algorithms. Section 4 describes LARA, our optimized NRA algorithm, which is built on these properties. In Section 5, we discuss important variants of top-k queries and how LARA can be adapted for each of them. LARA is experimentally compared with previous algorithms of NRA top-k search in Section 6. Finally, Section 7 concludes the paper.

2 Background and Related Work

Let D be a collection of n objects (e.g., images) and $S_1, S_2, \ldots, S_m$ be a set of m *ranked inputs* (e.g., search engine results) of the objects, based on their *atomic* scores (e.g., similarity to a query) on different features (e.g., color, texture, etc.). An aggregate function γ (e.g., weighted sum) maps the m atomic scores $x_1, x_2, \ldots, x_m$ of an object x in $S_1, S_2, \ldots, S_m$ to an aggregate score γ_x. Function γ is *monotone* if $(x_i \leq y_i, \forall i) \Rightarrow \gamma_x \leq \gamma_y$. Given γ, a *top-k query* on $S_1, S_2, \ldots, S_m$ (also called *rank aggregation*) retrieves R, a k-subset of D ($k < n$), such that $\forall x \in R, y \in D - R : \gamma_x \geq \gamma_y$. Consider the example of Figure 1 showing three ranked inputs for objects $\{a, b, c, d, e\}$ and assume that the score of an object in each source ranges from 0 to 1. A top-1 query with sum as aggregate function γ returns b with score $\gamma_b = \gamma(0.6, 0.8, 0.8) = 2.2$.

Fagin et al. [6] present a comprehensive analytical study of various methods for top-k aggregation of ranked inputs by monotone aggregate functions. They identify two types of accesses to the ranked lists; *sorted* accesses and *random* accesses. The first operation, iteratively reads objects and their scores sequentially, whereas a random access is a request for an object's score in some S_i given the object's ID. In some applications, both sorted and random accesses are possible, whereas in others, some of the sources may allow only sorted or random accesses.

For the case where sorted and random accesses are possible, a *threshold algorithm* (TA) (independently proposed in [6, 13, 7]) retrieves objects from the ranked inputs in a round-robin fashion and directly computes their aggregate scores by performing random accesses to the sources where the object has not been seen. A priority queue is used to organize the best k objects seen so far. Let l_i be the last score seen in source S_i; $T = \gamma(l_1, \ldots, l_m)$ defines a *threshold* (i.e., a lower bound) for the aggregate score of objects never seen in any S_i yet. If the k-th highest aggregate score found so far is at least equal to T, then the algorithm is guaranteed to have found the top-k objects and terminates.

S_1	S_2	S_3
c 0.9	a 0.9	c 0.9
d 0.8	b 0.8	a 0.9
b 0.6	e 0.6	b 0.8
e 0.3	d 0.4	d 0.6
a 0.1	c 0.2	e 0.5

Figure 1. Three ranked inputs

For the case where random accesses are either impossible or much more expensive compared to sorted ones, [6] proposes an algorithm, referred to as "no-random accesses" (NRA). NRA computes the top-k result, performing sorted accesses only. It iteratively retrieves objects x from the ranked inputs in a round-robin fashion. NRA maintains these objects and upper γ_x^{ub} and lower γ_x^{lb} bounds of their aggregate scores, based on their atomic scores seen so far and the upper and lower bounds of scores in each S_i where they have not been seen. Bound γ_x^{ub} is computed by assuming that for every S_i, where x has not been seen yet, x's score in S_i is the highest possible (i.e., the score l_i of the last object seen in S_i). Bound γ_x^{lb} is computed by assuming that for every S_i, where x has not been seen yet, x's score in S_i is the lowest possible (i.e., 0 if scores range from 0 to 1). Let W_k be the set of the k objects with the largest γ^{lb}. If the smallest lower bound in W_k is at least the largest γ_x^{ub} of any object x not in W_k, then W_k is reported as the top-k result and the algorithm terminates. NRA is described by the pseudocode of Figure 2.

Algorithm NRA(ranked inputs $S_1, S_2, \ldots, S_m$)
1. perform a sorted access on each S_i;
2. **for** each newly accessed object x update γ_x^{lb};
3. **if** less than k objects have been seen so far **then** goto Line 1;
4. **for** each object x seen so far compute γ_x^{ub};
5. W_k := the k objects with the highest γ^{lb};
6. $t := \min\{\gamma_x^{lb} : x \in W_k\}$;
7. $u := \max\{\gamma_x^{ub} : x \notin W_k\}$;
8. **if** $t < u$ **then** goto Line 1;
9. report W_k as the top-k result;

Figure 2. The NRA algorithm

Let us see how NRA processes the top-1 query for the ranked inputs of Figure 1 and $\gamma =$ sum, assuming that the

atomic scores in each source range from 0 to 1. In the first loop, NRA accesses c (from S_1 and S_3) and a (from S_2). $W_1 = \{c\}$, where $\gamma_c^{lb} = 1.8$. In addition, the object with the highest γ^{ub} is a, with $\gamma_a^{ub} = 2.7$. Since $\gamma_c^{lb} < \gamma_a^{ub}$, NRA loops to access a new round of objects (Line 8). After a second and a third round of accesses, $W_1 = \{b\}$, with $\gamma_b^{lb} = 2.2$ (which happens to be the exact score γ_b of b, since we saw it in all sources). NRA still does not terminate, because $\gamma_c^{ub} = 2.4 > \gamma_b^{lb}$ (i.e., c may eventually become the best object). After the fourth round, $W_k = \{b\}$ and the highest upper bound is $\gamma_c^{ub} = 2.2$. Since $\gamma_c^{ub} \leq \gamma_b$ the algorithm terminates, reporting b as the top-1 object.

A simple variation of the basic NRA algorithm is Stream-Combine (SC) [8]. SC reports only objects which have been seen in all sources, thus their scores should be exact and above the best-case score of all objects not in W_k. In addition, an object is reported as soon as it is guaranteed to be in the top-k set. In other words, the algorithm does not wait until the whole top-k result has been computed in order to output it, but provides the top-k objects with their scores *on-line*. A difference of SC with NRA is that it does not maintain W_k, but only the top-k objects with the highest γ^{ub}. If one of these objects has its exact score computed, it is immediately output.

J* is a more generic rank aggregation operator proposed in [12]. J* is appropriate for merging ranked inputs based on a join condition on attributes other than the scores, such that only the k join results with the highest aggregate scores are output. The top-k query we have defined is a special case of this problem, where the joined attributes are the object IDs (unique in each S_i) and the join condition is equality. J* can be used as an operator in a query plan which joins multiple ranked inputs. Nevertheless, for top-k queries J* is less efficient than NRA, as shown in [9].

NRA-RJ [9] is a 'partially' non-blocking version of NRA, which outputs an object as soon as it is guaranteed to be in the top-k (like SC), however, without necessarily having computed its exact aggregate score (like NRA). This may affect the operability of following operators, if they require exact aggregate scores or values of other attributes than the score. In view of this, [10] proposed another version of NRA that outputs exact scores on-line (like SC) and can be applied for any join predicate (like J*). This algorithm uses a threshold which is inexpensive to compute, appropriate for generic rank join predicates. However, it is much looser compared to T and incurs more object accesses than necessary in top-k queries. [12], [9], and [10] focused on binary implementations of top-k join algorithms, which can be used as operators in queries that involve ranking. However, as already shown in [9], non-binary algorithms could be more efficient than combinations of binary operators for problems with $m > 2$. In this paper, we propose an efficient non-binary operator for top-k queries which can also be adapted for generic rank joins.

Finally, [2, 3] study top-k queries in environments where the scores of the objects can be accessed sequentially from only one source, whereas the other sources allow (possibly expensive) random score evaluations. Adapted versions of TA were proposed for this case. Besides, probabilistic extensions of top-k algorithms for approximate retrieval have been proposed in [14].

3 The Two Phases of NRA Methods

In this section, we motivate our research and bring to light some key observations on the behavior of "no random accesses" (NRA) top-k algorithms. These observations impose two phases that any NRA algorithm should essentially go through; a *growing* and a *shrinking* phase.

3.1 Motivation

NRA (see Figure 2) repeatedly accesses objects from the sorted inputs, updates the worst-case and best-case scores of all objects seen so far, and checks whether the termination condition holds. Note that, from these operations, updating γ_x^{lb} and W_k can be performed fast. First, only a few objects x are seen at each loop and γ_x^{lb} should be updated only for them. Second, the k highest such scores can be maintained in W_k efficiently with the help of a priority queue. On the other hand, updating γ_x^{ub} for each object x is the most time-consuming task of NRA. Let l_i be the last score seen so far in S_i. When a new object is accessed from S_i, l_i is likely to change. This change affects the upper bounds γ_x^{ub} for all objects that have been seen in some other stream, but not S_i. Thus, a significant percentage of the accessed objects must update their γ_x^{ub}; it is crucial to perform these updates efficiently and only when necessary.

Another important issue is the minimization of the required memory, i.e., the maximum number of candidate top-k objects. NRA (see Figure 2) allocates memory for every newly seen object, until the termination condition $t \geq u$ is met. However, during top-k processing, we should avoid maintaining information about objects that we know that may never be included in the result. Finally, we should avoid redundant accesses to any input S_i that does not contribute to the scores of objects that may end up in the top-k result.

3.2 Behavior of NRA algorithms

We now provide a set of claims that impose some useful rules towards defining a top-k algorithm of minimal computational cost, memory requirements, and object accesses. Let t be the k-th highest score in W_k and $T = \gamma(l_1, \ldots, l_m)$. We can show the following:

Claim 1 If $t < T$, objects which have not been seen so far

in any input can end up in the top-k result.

Proof. Let y be the k-th object in W_k and x be an object, which has not been seen so far in any S_i. The score of y in all inputs where y has not been seen, could be the lowest possible, that is $\gamma_y = \gamma_y^{lb} = t$. In addition, the atomic scores of x could be the highest possible, i.e., $x_i = l_i$ in all inputs S_i, resulting in $\gamma_x = T$. $t < T$ implies that we can have $\gamma_y < \gamma_x$, thus x can take the place of y in the top-k result. □

Claim 2 If $t < T$, any of the objects seen so far can end up in the top-k result.

Proof. Let y be the k-th object in W_k and x be an object which has been seen in at least one input. If $x \in W_k$ the claim trivially holds. Let $x \notin W_k$. From the monotonicity property of γ, we can derive that $T \leq \gamma_x^{ub}$, since in the sources S_i, where x has been seen, x's score is at least l_i and in all other inputs S_j, x's score can be l_j in the best case. From $\gamma_y^{lb} = t < T$ and $T \leq \gamma_x^{ub}$, we get $\gamma_y^{lb} < \gamma_x^{ub}$, which implies that x can replace y in the top-k result. □

Claims 1 and 2 imply that while $t < T$ the set of candidate objects can only *grow* and there is nothing that we can do about it. Thus, while $t < T$, *we should only update W_k and T while accessing objects from the sources and need not apply expensive updates and comparisons on γ_x^{ub} upper bounds*.

As soon as $t \geq T$ holds, NRA should start maintaining upper bounds and compare the highest γ_x^{ub} ($\forall x \notin W_k$) with t, in order to verify the termination condition of Line 8 in Figure 2. An important observation is that if $t \geq T$, all objects that have never been seen in any S_i cannot end up in the top-k result:

Claim 3 If $t \geq T$, no object which has not been seen in any input can end up in the top-k result.

Proof. Let y be the k-th object in W_k and x be an object, which has not been seen so far in any S_i. Then $\gamma_x \leq T$, because $x_i \leq l_i, \forall i$ and due to the monotonicity of γ. Thus $\gamma_x \leq T \leq t \leq \gamma_y^{lb} \leq \gamma_y$, i.e., the aggregate score of x cannot exceed the aggregate score of y. □

The implication of Claim 3 is that once condition $t \geq T$ is satisfied, the memory required by the algorithm can only shrink, as we need not keep objects never been seen before. Summarizing, Claims 1 through 3 imply two phases that all NRA algorithms go through; a *growing* phase during which $T < t$ and the set of top-k candidates can only grow and a *shrinking* phase during which $t \geq T$ and the set of candidate objects can only shrink, until the top-k result is finalized. Finally, the next corollary (due to Claim 3) helps reducing the accesses during the shrinking phase.

Corollary 1 If $t \geq T$ and all current candidate objects have already been seen accessed from input S_i, no further accesses to S_i are required in order to compute the top-k result.

4 Lattice-based Rank Aggregation

Our Lattice-based Rank Aggregation (LARA) algorithm is an optimized "no random accesses" method, based on the observations discussed in the previous section. We identify the operations required in each (growing and shrinking) phase and choose appropriate data structures, in order to support them efficiently. LARA takes its name from the lattice it uses to reduce the computational cost and the number of sorted accesses in the shrinking phase. For now, we will assume that the aggregate function γ is (weighted) sum. Later, we will discuss the evaluation of top-k queries that involve other aggregate functions, as well as combinations thereof.

4.1 The growing phase

As discussed, while $t < T$ (i.e., during the growing phase), the set of candidate objects can only grow and it is pointless to attempt any pruning. Thus LARA only maintains (i) the set of objects seen so far with their partial aggregate scores[1] and the set of sources where from each object has been accessed, (ii) W_k, the set of top-k objects with the highest lower score bounds (used to compute t), and (iii) an array L with the highest scores seen so far from each source (used to incrementally compute T).

We implement (i) by a hash table H (with object-ID as search key) that stores, for each already seen object x, its ID, a bitmap indicating the sources where from x has been seen, its aggregate score γ_x^{lb} so far, and a number pos_x (to be discussed shortly). For (ii), we use a heap (i.e., priority queue) to organize W_k. Whenever an object x is accessed from an input S_i, we update the hash table with γ_x^{lb} (in O(1) time). At the same time, we check if x is already in W_k. For this, we use entry pos_x, which denotes the position of x in the heap of W_k (pos_x is set to $k+1$ if x is not in W_k). If x already existed in W_k, its position is updated in W_k (in O($\log k$) time) and the O($\log k$) positional updates of any other object in W_k are reflected in the hash table (in O(1) time for each affected object). If x is not in W_k, its updated γ_x^{lb} is compared to that of the k-th object in W_k (i.e., the current value of t) and, if larger, a replacement takes place (again in O($\log k$) time). Finally, L is updated and T is incrementally computed from the previous value in O(1) time; if $\gamma = \texttt{sum}$ and T^{prev} (l_i^{prev}) denotes the value of T (l_i) before the last access, then $T = T^{prev} - l_i^{prev} + l_i$.

After each access, the data structures are updated and the condition $t \geq T$ is checked. The first time this condition is true, the algorithm enters the shrinking phase, discussed in the next paragraph. The overall time required to update

[1]The *partial aggregate score* is (incrementally) derived when γ is applied only on the set of inputs where x has been seen. If γ is (weighted) sum then this score corresponds to γ_x^{lb}.

the data structures and check the condition $t \geq T$ for advancing to the shrinking phase is O($\log k$) per access, which is worst-case optimal given the operations required at this phase.

4.2 The shrinking phase

Once $t \geq T$ is satisfied, LARA progresses to the shrinking phase, where upper score bounds are maintained and compared to t, until the top-k result is finalized. LARA applies several optimizations, in order to improve the performance of this phase.

4.2.1 Immediate pruning of unseen objects

According to Claim 3, during the shrinking phase, no new objects can end up in the top-k query result; if a newly accessed object is not found in the hash table, it is simply ignored and we proceed to the next access. This not only saves many unnecessary computations, but also reduces the memory requirements to the minimal value (i.e., the number of accessed objects until $t \geq T$); no more memory will ever be needed by the algorithm.

4.2.2 Efficient verification of termination

Let C be the set of candidate objects that can end up in the top-k result. Let x be the object in $(C - W_k)$ with the greatest γ_x^{ub}; the algorithm terminates if $\gamma_x^{ub} \leq t$. An important issue is how to efficiently maintain γ_x^{ub}. A brute-force technique (to our knowledge, used by previous NRA implementations [6, 8, 9]) is to explicitly update γ^{ub} for all objects in C and recompute γ_x^{ub}, after each access (or after a number of accesses from each source). This involves a great deal of computations, since all objects must be accessed and updated.

Instead of explicitly maintaining γ_x^{ub} for each $x \in C$, LARA reduces the computations based on the following idea. For every combination v in the powerset of m inputs $\{S_1, \ldots, S_m\}$, we keep track of the object x^v in C such that (i) x^v has been seen exactly in the v inputs, (ii) $x^v \notin W_k$, and (iii) x^v has the highest partial aggregate score among all objects that satisfy (i) and (ii). Note that if $\gamma_{x^v}^{ub} \leq t$ we can immediately conclude that no candidate seen exactly in the v inputs may end up in the result. Thus, by maintaining the set of x^v objects, one for each combination v, we can check the termination condition by performing only a small number[2] of $O(2^m)$ comparisons.

Specifically, as soon as LARA enters the shrinking phase, it constructs a (virtual) lattice $\mathcal{G}$. For every combination v of inputs (i.e., node in $\mathcal{G}$), it maintains the ID of its *leader* x^v, which is the object with the highest partial aggregate score seen only in v, but currently not in W_k. If t is not smaller than any $\gamma_{x^v}^{ub}$ for each v, LARA terminates reporting W_k.

Let us now discuss how the data structures maintained by LARA are updated after a new object x has been accessed from an input S_i. One of the following cases apply, after x is looked up in the hash table H:

1. x is not found in H. In this case, x is ignored, as discussed in paragraph 4.2.1.
2. $x \in W_k$ (checked by pos_x). In this case, γ_x^{lb} is updated and so is x's position in the priority queue of W_k.
3. $x \notin W_k$. In this case, we first check whether x was the leader of the lattice node v_x^{prev} where x belonged, before it was accessed at S_i. If so, a new leader for v_x^{prev} is selected. Then, we check whether x can now enter W_k (by comparing it with t^{prev}). If so, we check whether the object evicted from W_k becomes a leader for its corresponding lattice node. Otherwise, x is *promoted* from v_x^{prev} to the parent node which contains S_i in addition to the other inputs, where x has been seen (and we check whether it becomes the new leader there).

4.3 The basic version of LARA

LARA, as presented so far, is described by the pseudocode of Figure 3. The algorithm repeatedly accesses objects from the various inputs and depending on whether it is in the growing or shrinking phase it performs the appropriate operations. As an example of LARA's operability, consider again the top-1 query on the three inputs of Figure 1, for $\gamma = \texttt{sum}$. Let us assume that the inputs are accessed in a round-robin fashion. After three rounds of sorted accesses (9 accesses), LARA enters the shrinking phase, since $t = \gamma_b^{lb} = 2.2$ and $T = 0.6 + 0.6 + 0.8 = 2.0$. Figure 4a shows the contents of the lattice, W_k ($k = 1$), and $L = \{l_1, l_2, l_3\}$ at this stage. For instance, object c (assigned to node S_1S_3, where it is also the leader) has been seen at exactly S_1 and S_3. c's score considering only these dimensions is 1.8. To compute $\gamma_{x^{S_1S_3}}^{ub} = \gamma_c^{ub}$, LARA adds l_2 (the highest possible score c can have in S_2) to γ_c^{lb}. Since $\gamma_{x^{S_1S_3}}^{ub} > t$, LARA proceeds to access the next object from S_1, which is e. Now, γ_e^{lb} becomes $0.9 < t$ and the object is promoted to node S_1S_2. We still have $\gamma_{x^{S_1S_3}}^{ub} > t$, thus LARA accesses the next object from S_2, which is d. Now, γ_d^{lb} becomes $1.2 < t$ and the object is promoted to S_1S_2. Figure 4b shows the lattice at this stage. Note that now $\gamma_{x^v}^{ub}$ for every (occupied) lattice node is at most t (i.e., $\gamma_{x^{S_1S_2}}^{ub} = \gamma_d^{ub} = 2.0, \gamma_{x^{S_1S_3}}^{ub} = \gamma_c^{ub} = 2.2, \gamma_{x^{S_2S_3}}^{ub} = \gamma_a^{ub} = 2.1$), thus LARA terminates.

Note that no objects can be assigned to the bottom $\varnothing$ and top $S_1 \ldots S_m$ nodes of the lattice. $\varnothing$ virtually contains all

[2] Top-k queries usually combine a small $m \leq 10$ number of ranked inputs [5]. Thus, in typical applications, $n \gg 2^m$

Algorithm LARA(ranked inputs $S_1, S_2, \ldots, S_m$)
1. $growing$:= `true`; /* initially in growing phase */
2. access next object x from next input S_i;
3. **if** $growing$ **then**
4.   update γ_x^{lb}; /* partial aggr. score */
5.   **if** $\gamma_x^{lb} > t$ **then**
6.   update W_k to include x in the correct position;
7.   update T;
8.   **if** $t \geq T$ **then**
9. $growing$:= `false`; construct lattice;
10.   goto Line 2;
11. **else** /* shrinking phase */
12.   **if** x in H **then**
13.   update γ_x^{lb}; /* partial aggr. score */
14.   **if** $x \in W_k$ **then** /* already in W_k */
15.   update W_k to include x in the correct position;
16.   **else** /* x was not in W_k */
17. v_x^{prev} := lattice node where x belonged;
18.   **if** x was leader in v_x^{prev} **then**
19.   update leader for v_x^{prev};
20.   **if** $\gamma_x^{lb} > t$ **then**
21.   update W_k to include x in the correct position;
22.   check if y (evicted from W_k) is leader of v_y;
23.   **else** check if x is leader of node $v_x := v_x^{prev} \cup S_i$;
24. $u := \max\{\gamma_{x^v}^{ub} : v \in \mathcal{G}\}$; /* use lattice leaders */
25.   **if** $t < u$ **then** goto Line 2;
26. report W_k as the top-k result;

Figure 3. The LARA algorithm

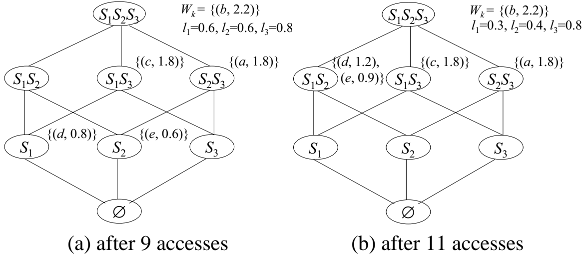


Figure 4. The lattice at two stages of LARA

(useless) objects never been seen during the growing phase and $S_1 \ldots S_m$ contains objects seen at all sources. None of the objects seen at all sources can be further improved; $\gamma^{ub} = \gamma^{lb}$ for them. Thus these are either in W_k, or pruned.

4.4 Analysis and Optimizations

Each access in the shrinking phase of LARA may involve (i) updating W_k, (ii) updating the leader of the lattice node where x existed, and/or (iii) updating the leader of the lattice node where to x is promoted. Operation (i) costs O($\log k$) time (as in the growing phase). The cost of (ii) is O(n), since we need to scan the entire set of candidates in order to find the new leader. Nevertheless, (ii) is not required unless x used to be the leader of v_x^{prev}, which happens with expected probability $\frac{1}{|v_x^{prev}|}$. Thus, the amortized cost of operation (ii) is O($\frac{n}{|v_x^{prev}|}$). Quantity $|v_x^{prev}|$ is n in the worst case, but its expected value is $\frac{n}{2^m}$. Hence, the cost of (ii) is expected to be O(2^m). Finally, the cost of (iii) is O(1), since a mere comparison to the previous leader is required. Summing up, for each access in the shrinking phase, the lattice maintenance cost for LARA is O($\log k + 2^m$) and checking the termination condition requires O(2^m) comparisons (as discussed). Overall, the cost of LARA (at each access) is O($\log k$) in the growing phase and O($\log k + 2^m$) in the shrinking phase. These numbers are much lower compared to the O(n) cost of NRA (assuming $n \gg 2^m$). Our experimental results verify the performance gap between LARA and past NRA algorithms. In the next paragraphs, we discuss some optimizations that further reduce the computational cost and the number of object accesses during the shrinking phase of LARA.

4.4.1 Reducing the number of candidates

The basic version of LARA (see Figure 3), does not explicitly prune any object, but keeps updating their lower and upper bounds until the termination condition holds. However, we can reduce the number of top-k candidates at minimal cost, during the regular operations of LARA. First, if for the last accessed object x, $\gamma_x^{ub} \leq t$, we can immediately delete x from H and avoid its promotion to the parent lattice node v_x. Consider again the application of LARA on the example of Figure 1, right after the 9th access (Figure 4a). When e is accessed from S_1 (10th access), γ_e^{ub} becomes $0.9 + 0.8 < 2.2 = t$, thus LARA can immediately prune e and avoid promoting it to node S_1S_2.

As a second optimization, during the execution of the algorithm, if all objects in a lattice node v have γ^{ub} not greater than t (verified by comparing t with the leader x_v of v), we can safely prune all objects from v, significantly reducing the number of candidates in H and avoiding redundant update operations (for these objects) in the future.

4.4.2 Reducing the number of comparisons

At the beginning of the shrinking phase, the majority of the lattice nodes are populated and highly unlikely to be pruned, since t is marginally greater than T, and as a result much smaller than the upper score bounds of most objects. Since we expect that the comparisons right at the beginning of the shrinking phase will hardly prune any object or node, it is wise to *delay* pruning attempts until there are high chances for the termination condition to hold.

Let u be the largest upper bound of objects not in W_k, when LARA enters the shrinking phase (i.e., $u := \max\{\gamma_{x^v}^{ub} : v \in \mathcal{G}\}$). If $u < t$, LARA immediately terminates (Lines 25–26 of Figure 3). Every new access (e.g., from source S_i) reduces $\gamma_{x^v}^{ub}$ for half of the lattice nodes (e.g., those including S_i) by $\Delta l = l_i^{prev} - l_i$, where l_i^{prev} is the previous value in S_i (before l_i was accessed). In addition, the new access might increase t. However, note that it

is not possible to prune all lattice nodes, while $u - \Delta l > t$. Thus, after computing u for the first time, and after every consequent access, instead of updating all upper bounds and performing lattice operations, while $u - \Delta l > t$, we set $u := u - \Delta l$ (and update t as usual), without attempting any actual comparisons. As soon as $u - \Delta l \leq t$, we begin updating upper bounds (and u).

4.4.3 Reducing the number of accesses

At the latter stages of LARA, we can exploit Corollary 1 to avoid accessing inputs that do not contribute to the aggregate score of remaining candidates. Let S_i be a source (e.g., S_1), such that (i) all objects in W_k have already been seen at S_i and (ii) for all lattice nodes v that do not contain that source (e.g., S_2, S_3, S_2S_3) $\gamma_{x^v}^{ub} \leq t$. Obviously, no object in any of these nodes can end up in the top-k result. In addition, for all objects x in all other nodes (e.g., S_1, S_1S_2, S_1S_3, $S_1S_2S_3$), x_i is already known. Thus, *no more accesses to S_i are needed* for computing the top-k result.

Based on this idea, LARA, while checking for the termination condition, keeps track of the pruned/empty nodes, whose subsets are all pruned/empty (i.e., by the use of a bitmap). In addition, it maintains a bitmap b_{W_k} which indicates the sources where all objects in W_k have been seen. The termination condition is checked in a level-wise bottom-up fashion, starting from nodes with one input S_i, then moving to nodes with two inputs S_iS_j, etc. At the first level, all pruned/empty nodes which are also set in b_{W_k} are marked as 'dead'. At level l a node is marked 'dead' only if (i) the node is pruned/empty, (ii) its immediate subsets are all marked 'dead', and (iii) the corresponding combination of bits in b_{W_k} is set. A dead node v needs never been checked again in the future, since, there may be no new object x that can end up in v with $\gamma_x^{ub} > t$. We can 'dry up' inputs by exploiting Corollary 1 as follows. Let $v = S_1S_2 \ldots S_{i-1}S_{i+1} \ldots S_m$ be a lattice node that contains all nodes but S_i. If v is marked dead, then we know that it is pointless to attempt any more accesses from S_i. S_i is then *dried up* and the total number of accesses is decreased.

Note that if LARA follows the same read schedule as NRA [6], it never performs more accesses than NRA. Thus, LARA is *instance optimal* [6] with respect to the number of performed accesses. On the other hand, LARA may perform fewer accesses than NRA in the possible case that an input S_i has been dried up before the top-k result has been finalized, by simply rejecting accesses to S_i from the read schedule.

In summary, LARA achieves (i) high computational efficiency by attempting no pruning during the growing phase and exploiting the lattice and several optimizations in the shrinking phase, (ii) minimal memory requirements by ignoring any new object once $T \geq t$, and (iii) minimal object accesses by 'drying up' inputs.

5 Variants of Top-k Search

So far we have discussed how LARA processes top-k queries when $\gamma = \texttt{sum}$. In addition, note that LARA can terminate before the complete scores of all objects in the top-k result are known. Finally, the algorithm does not output any result until the whole top-k set is known and does not *incrementally* output the results in increasing order of their aggregate scores. In this section, we show how LARA can be adapted for different variants and requirements of a top-k search.

5.1 Exact scores

LARA terminates as soon as $t \geq \gamma_x^{ub}, \forall x \notin W_k$, even when the exact aggregate score is not known for all objects in W_k. Some applications, however, may require the exact scores of all objects in the top-k result. LARA can be easily adapted to address this requirement, at the probable expense of performing additional accesses. LARA-EX operates exactly like the original algorithm, but after the top-k result has been finalized, it continues to perform accesses to the inputs, where each $x \in W_k$ has not been seen yet, until the exact scores of all objects have been computed. Another difference between LARA-EX and the basic algorithm is that after entering the shrinking phase, the lattice is not directly constructed, but we continue to access objects (disregarding those that were not seen in the growing phase), until W_k contains only objects that have been seen at all inputs. Only then the lattice is constructed and pruning begins.

5.2 On-line and incremental search

Some applications require *on-line* generation of the top-k result; as soon as an object is guaranteed to be in the top-k result, it is immediately output. We propose an adaptation of LARA, denoted by LARA-OL, that serves this purpose. In the growing phase, LARA-OL, maintains, apart from W_k, the object h with the highest γ^{lb}. As long as $\gamma_h^{lb} < T$, we know that h is not guaranteed to be better than all unseen objects (due to Claim 1). When $\gamma_h^{lb} \geq T$, LARA-OL constructs the lattice and compares upper bounds with γ_h^{lb}, until $\gamma_h^{lb} \geq \gamma_{x^v}^{ub}, \forall v \in \mathcal{G}$. When this condition becomes true, h is immediately output, since we can be certain that it is part of the top-k result. At this point, we decrement $k := k - 1$ and compute the new h. If $\gamma_h^{lb} < T$, the algorithm again enters the phase of accessing objects and updating W_k and h, until $\gamma_h^{lb} \geq T$; then the lattice is re-constructed and upper-bound comparisons begin. A subtle matter to note is that W_k is maintained during the whole process for the purpose of comparing t with T. As soon as $t \geq T$, never-seen objects can be ignored.

LARA-OL can be easily converted to LARA-IN; an algorithm that outputs the objects with the highest score *incrementally*, without a constraint k. LARA-IN does not maintain W_k, but only the object h with the highest score in the worst case (i.e., with the highest γ^{lb}). As long as $\gamma_h^{lb} < T$ we simply access objects and maintain h. When $\gamma_h^{lb} \geq T$, the lattice is constructed and maintained, until $\gamma_h^{lb} \geq \gamma_{x^v}^{ub}, \forall v \in \mathcal{G}$; h is then output as the next object. Note that since LARA-IN is incremental, no object is ever pruned; LARA-IN outputs *all* objects in decreasing order of their aggregate score. Thus, all new objects are considered for inclusion in the lattice. After h has been output the new h is computed and if $\gamma_h^{lb} < T$ the algorithm again enters the phase of merely accessing objects and updating h, until $\gamma_h^{lb} \geq T$; then the lattice is updated and reused.

LARA-IN can be implemented as a (binary or multiway) top-k operator in a complex query processing plan that may also involve other operators (i.e., like the NRA-RJ operator proposed in [9]). For demand-driven retrieval, a *GetNext*() function accesses object scores from the input sources (which may be tuples also containing other attributes) and produces h when it is guaranteed to have the highest aggregate score. After producing h, the LARA-IN operator maintains its state, from which it continues at the next call of *GetNext*().

5.3 Top-k *join* queries

The top-k query we have seen so far is a special case of top-k *join* queries [12, 10, 11], where the results of joins are to be output in order of an aggregate score on their various components. Consider, for example, the following top-k query expressed in SQL:

```
SELECT R.id, S.id, T.id
FROM R, S, T
WHERE R.a = S.a
  AND S.b = T.b
ORDER BY R.score + S.score + T.score
STOP AFTER k;
```

The top-k query we have examined is a special case, where `id = a = b`, tuple `ids` are unique, all `R`, `S`, `T` have the same collection of tuple `ids`, and tuples from each relation are ranked based on their scores. [12, 10] propose algorithms for solving generic top-k joins. Here, we discuss how LARA can be converted to LARA-J, a top-k join operator that incrementally outputs join results based on their aggregate scores. Instead of maintaining a single hash table with all objects seen so far, LARA-J materializes the lattice and stores for each combination of sources, tuple combinations that partially satisfy the join (e.g., tuples from `R` that match with tuples from `S` are stored in node $\texttt{R} \circ \texttt{S}$ of the lattice). For each lattice node, the combination with the highest γ^{ub} is maintained as usual. LARA-J does not keep a W_k, but combinations in the top lattice node (e.g., $\texttt{R} \circ \texttt{S} \circ \texttt{T}$) are organized in a priority queue based on their aggregate score. These are output incrementally, as soon as they are known to have greatest score than all γ^{ub} in the rest of lattice nodes. When a new tuple is read (e.g., from `R`), it is immediately joined with combinations in the lattice nodes that do not include its source, but include sources joined with it (e.g., nodes `S` and $\texttt{S} \circ \texttt{T}$, but not `T`). The new tuple is accommodated in the corresponding lattice node and join results are immediately added to the corresponding nodes. Combinations in each lattice node are indexed in order to facilitate efficient probing (e.g., using hash tables).

5.4 Other aggregate functions

We now discuss how LARA (and NRA top-k methods in general) can be adapted to solve top-k queries with aggregate functions other than `sum` and combinations of them. A trivial function is `max`; a top-k `max` query can be processed by accessing at most k objects from each input, which guarantees that the k objects with the maximum score in *any* input are found.

5.4.1 The `min` aggregate function

A function which requires special attention is `min`; a top-k `min` query asks for the k objects with the highest *minimum* score at all inputs. Without loss of generality, let us assume that the minimum possible score at each input is 0. In that case, γ_x^{lb} is 0 for all objects which have not been seen at all inputs. As a result, the growing phase terminates when k objects have been seen at all inputs. When this happens, the score of the last object in W_k is at least the smallest score seen in any input (i.e., $t \geq T$). Thus, when $\gamma = \texttt{min}$, only exact (not partial) scores can be output.

In the shrinking phase, accessing objects from any S_i where $l_i \leq t$ is of no use, since no object which has not been seen there can end up in the top-k result. When LARA enters the shrinking phase, it immediately prunes all lattice nodes (and their objects) that do not include any of these streams and 'dries up' the streams. These operations are encompassed by the optimization of Section 4.4.3.

We can further improve the efficiency of LARA by delaying the beginning of the shrinking phase as follows. Instead of accessing the inputs in a round-robin fashion, we always expand the input with the largest l_i. By doing so, the number of objects with γ^{ub} greater than t, when the shrinking phase begins, will be minimized, since their maximum potential scores in the inputs where they have not been seen will not be much greater than t.

5.4.2 Weighted and complex aggregates

So far, we have discussed top-k queries for which all components have equal weights. In practice, the user may assign

weights of importance to each input of the aggregate function. For example, assuming that $m = 3$, a *weighted* sum function could be defined as $\gamma_x = 0.5x_1 + 0.3x_2 + 0.2x_3$, indicating that the importance of S_1 (50%) is greater than the importances of S_2 (30%) and S_3 (20%) in the merged scores. Similarly, weight coefficients can be combined with other aggregate functions, like min. LARA can be directly applied for weighted functions. A simple optimization is to access inputs of higher weight with higher probability, as they contribute more to the aggregate function. In this way, objects which have not been seen in the sources of higher weights will be pruned earlier, resulting at an early termination of the algorithm.

In general, an aggregate function can be defined by a regular expression involving nested (potentially weighted) sum, min, max subexpressions. An example of such a function is $\gamma = \texttt{min}\{x_1, \texttt{sum}\{0.5x_2, 0.5x_3\}\}$. An interesting issue is whether we can extend top-k algorithms to process such complex functions. A plausible solution is to use binary, incremental top-k operators in a query evaluation tree, as suggested in [12, 9, 10, 11]. Another possibility is to process all inputs simultaneously by a single application of a top-k algorithm. In this case, lower and upper bounds are defined for an object by applying the complex aggregate function using the values seen so far and the minimum and maximum values at the inputs, where the object has not been seen. LARA can directly be applied for such complex aggregate functions.

6 Experimental Evaluation

In this section, we experimentally evaluate the effectiveness of LARA, by comparing it with previous NRA algorithms. For each top-k variant, (i.e., classic top-k search, exact scores, incremental search, etc.), a version of LARA is compared with a version of NRA known to perform best for that variant. All algorithms were implemented in C++ and experiments were run on a Pentium 4 2.26GHz PC with 512 MB of RAM.

6.1 Description of datasets

For the experiments, we used both synthetically generated and real data. All generated object scores range from 0 to 1. We produced three types of synthetic datasets to model different input scenarios, using the same methodology as [1]. In datasets of type UI the object scores are random numbers uniformly and independently generated for the different sources. CO contains datasets where object scores are *correlated*. In other words, the score x_i of an object x in source S_i is very close to x_j in all other sources $S_j \neq S_i$ with high probability. An real dataset example that falls in this class is a set of movies with their scores according to different criteria (actors performance, costumes design, visual effects, etc.). A good movie is likely to have high scores in all criteria, whereas a B-movie is likely to perform averagely or bad in all of them. To generate an object x, first, a number μ_x from 0 to 1 is selected using a Gaussian distribution centered at 0.5. x's atomic scores are then generated by a Gaussian distribution centered at μ_x. Finally, AC contains datasets where object scores are *anti-correlated*. In this case, objects that are good in one dimension are bad in one or all other dimensions. For instance, a good hotel in terms of quality (e.g., 5-star) is usually a bad one in terms of price (e.g., very expensive) and vice versa. To generate an object x, first, we pick a number μ_x from 0 to 1, like we did for CO datasets. This time, however, we use a very small variance, so that μ_x for different x are very close to 0.5 and to each other. The atomic scores of x are then generated uniformly and normalized to sum up to μ_x. In this way, the aggregate scores of all objects are quite similar, but their individual scores vary significantly.

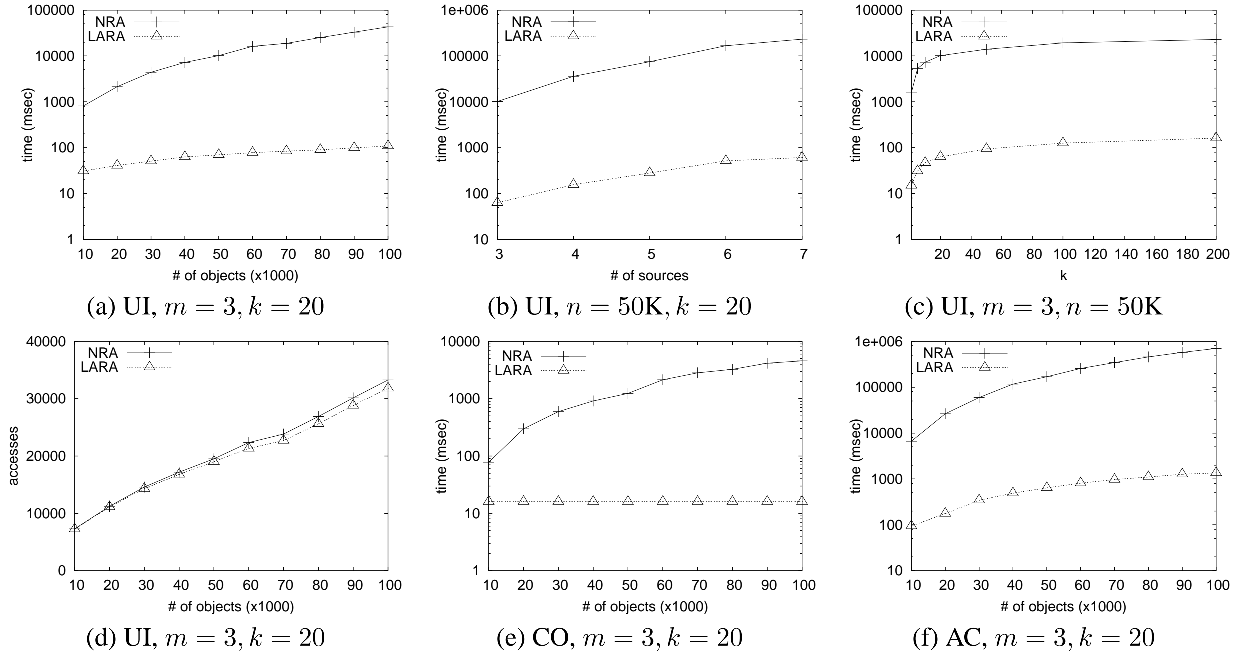
We used two real datasets from the UCI KDD Archive[3]. FC contains a set of 581,012 objects, corresponding to 30×30-meter forest land cells. Each object is described by various variables, such as horizontal and vertical distance to hydrology, distance to roadways, and distance to fire points. Assuming that the values of these attributes are obtained from different sources, we simulate top-k queries that combine them in an aggregate score (e.g., find the k cells with the smallest aggregate distance to hydrology, roadways, and fire points). The second dataset we used is CE, which contains unweighted Public Use Microdata Series (PUMS) census data from the Los Angeles and Long Beach areas for the years 1970, 1980, and 1990. Each object in CE is a person (or household) characterized by attributes such as age, rent, wages, number of working hours last week, and number of working weeks last year. Using combinations of these attributes, we can define aggregate ranking functions, like, e.g., life quality. The cardinality of CE is 84,443.

6.2 Experimental comparison

In the first set of experiments, we compare LARA with the NRA algorithm of [6] for top-k queries with $\gamma = \texttt{sum}$, in terms of object accesses and computational cost. We implemented both algorithms so that they check the termination condition after every access. In this way, the number of accesses is minimized, since every access in the shrinking phase can potentially terminate search.

Figures 5a–5c compare the efficiency (in CPU time) of the two methods on uniform data (UI), for a range of parameter values. The default values for the parameters are $n = 50$K, $m = 3$, and $k = 20$. In each experiment, we fix two parameters to their default values and vary the value of the third one. In all experimental instances, LARA is about 2

[3] http://kdd.ics.uci.edu



(a) UI, $m = 3, k = 20$ (b) UI, $n = 50\text{K}, k = 20$ (c) UI, $m = 3, n = 50\text{K}$

(d) UI, $m = 3, k = 20$ (e) CO, $m = 3, k = 20$ (f) AC, $m = 3, k = 20$

Figure 5. Top-k queries on synthetic data ($\gamma = \texttt{sum}$)

orders of magnitude faster than NRA. First, as explained in Section 4, LARA does not attempt checking for termination during the growing phase. Second, the shrinking phase of LARA is much more efficient than that of NRA, since at each access only a few updates are performed and the number of comparisons is O(2^m), as opposed to O(n) required by NRA. This explains the increase of performance gap with the increase of n. On the other hand, the difference is insensitive to m and k, as shown in Figure 5b and Figure 5c.

LARA outperforms NRA in terms of the number of object accesses, as well, but the difference is marginal. Indicatively, Figure 5d shows the number of object accesses on uniform data by both methods as a function of n. LARA's optimization described in Section 4.4.3 saves 1%–5% of NRA's accesses, because of 'dried up' streams towards the end of the algorithm. The difference in accesses is similar when other parameters change (i.e., m and k). As we will see later, LARA may accesses significantly fewer objects than NRA in top-k queries on real data, where the distribution of scores in different inputs varies significantly.

The performance gap between NRA and LARA is similar for correlated and anticorrelated data (Figures 5e and 5f). As expected, the cost of both methods is relatively low for correlated data, however, NRA becomes significantly more expensive than LARA for large n, since more operations (O(n)) are required by NRA, as n increases in both growing and shrinking phases. For AC, the cost is high (extreme for NRA), because many accesses are required until the top-k result is finalized. The shrinking phase delays and a lot of unnecessary bookkeeping is performed by NRA. Value n has a smoother effect on LARA, which manages to retrieve the result very fast compared to NRA.

In the next experiment, we compare LARA and NRA for top-k queries on real data. From the FC dataset we extracted four rankings of the objects according to their horizontal distance to hydrology (*hh*), vertical distance to hydrology (*vh*), horizontal distance to roadways (*hr*), and horizontal distance to fire points (*hf*). The distances in each ranking were normalized from 0 to 1 and reversed (by subtracting them from 1) in order for 1 to indicate high preference and 0 low preference. For different combinations of these rankings, we applied a top-20 query and compared the performances of LARA and NRA. Figure 6 summarizes the results. NRA performs 8% to 65% more accesses than LARA (see Figure 6a). This is attributed to the distribution of the scores which is irregular in some sources (e.g., *vh*); these sources are 'dried up' during the shrinking phase of LARA, when the remaining scores there cease to be relevant to the top-k result. On the other hand, NRA accesses objects in a round-robin fashion, without pruning any input, until the top-k result is finalized. Figure 6b shows that LARA is 3 to 4 orders of magnitute faster than NRA for the tested queries. Next to each time measurement, we also include in parentheses the time spent by each algorithm until $t \geq T$ for the first time. The numbers show that LARA is significantly faster than NRA not only because it avoids expensive book-

attributes of FC	NRA	LARA
{*hh, hr, hf*}	110529	102168
{*hh, vh, hr*}	358611	217054
{*hh, vh, hf*}	536097	393915
{*hh, hr, hf, vh*}	501231	315042

(a) number of accesses

attributes of FC	NRA	LARA
{*hh, hr, hf*}	517 (377)	0.25 (0.15)
{*hh, vh, hr*}	1047 (471)	0.5 (0.18)
{*hh, vh, hf*}	1614 (703)	0.8 (0.3)
{*hh, hr, hf, vh*}	3415 (1412)	1.06 (0.48)

(b) time in seconds

Figure 6. Forest coverage (γ =sum, $k = 20$)

attributes of CE	NRA	LARA
{*r, wh, wy, w*}	171187	161880
{*a, wh, wy, w*}	186694	185962
{*w, wh, wy*}	128390	107258
{*a, r, wh, wy, w*}	238182	237176

(a) number of accesses

attributes of CE	NRA	LARA
{*r, wh, wy, w*}	300 (227)	0.4 (0.25)
{*a, wh, wy, w*}	299 (228)	0.4 (0.25)
{*w, wh, wy*}	224(122)	0.25 (0.18)
{*a, r, wh, wy, w*}	432 (276)	0.56 (0.32)

(b) time in seconds

Figure 7. Census data (γ =sum, $k = 20$)

keeping and checking during the growing phase ($t < T$), but also because it minimizes the operations and comparisons at the shrinking phase ($t \geq T$).

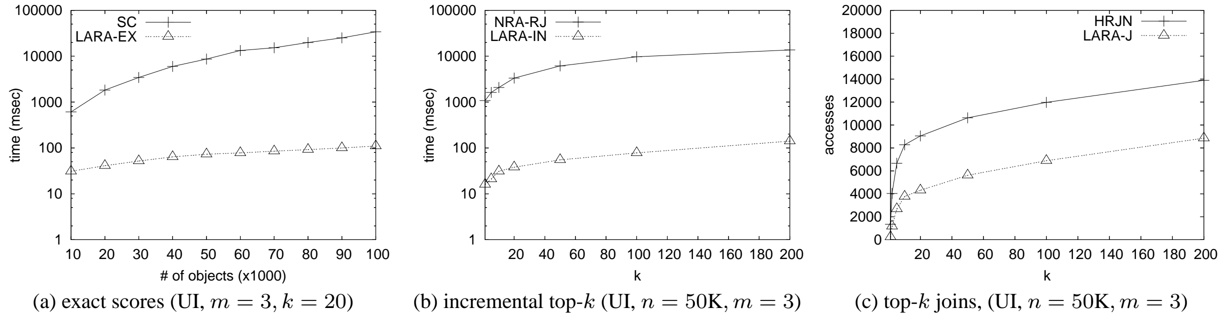
Figure 7 shows similar findings for top-20 queries on the CE dataset, for combinations of *r* (rent), *a* (age), *wh* (working hours per week), *wy* (working weeks per year), and *w* (wages) census data attributes. This time, the performance gap in accesses is not large, however, the improvement of LARA over NRA in terms of computations is huge. The efficiency of NRA can be improved if W_k and u are not updated/computed after every access, but every multiple accesses from each source. However, this increases the number of accesses. In addition, even when many (e.g., tens of) objects are accessed before checking for termination, LARA is still more than an order of magnitude faster than NRA. In summary, LARA minimizes the CPU cost of top-k queries, while keeping the number of accesses minimal.

In the next set of experiments we compare versions of LARA for variants of top-k processing with past NRA implementations for these variants. We first compare LARA-EX with SC [8] for exact top-k processing. In all tested cases with synthetic and real data, LARA-EX and SC have similar performances to LARA and NRA, respectively. Indicatively, here we show the performance as a function of n, for $m = 3$, $k = 20$ on uniform datasets (Figure 8a). Comparing with Figure 5a, observe that the performance of LARA-EX is very similar to that of LARA; when the top-k result is guaranteed to have been found, a few (or no) objects miss some score from W_k, so only a few (or no) extra accesses are required. SC is 5%-25% faster than NRA, because it avoids the maintenance of W_k; however, SC is still very expensive compared to LARA-EX.

We also implemented LARA-IN and NRA-RJ [9] and compared the two incremental algorithms. Note that LARA-IN and NRA-RJ do not prune any object, as all of them are incrementally output. In addition, the optimization of Section 4.4.3 is not applicable for LARA, since no stream can be pruned in incremental processing. Thus, the two algorithms perform exactly the same number of accesses (assuming that they follow identical access schedules).[4] Figure 8b shows their performances for uniform data as a function of the number of results. LARA-IN produces results at significantly higher speed compared to NRA-RJ, thus our algorithm is appropriate for top-k processing over fast streams. Results on other distributions and real data are similar and they are omitted due to space constraints.

Next, we compare the top-k *join* version of LARA (i.e., LARA-J) with a tree of binary HRJN operators [10]. We joined three relations `R`, `S`, and `T`, of the same schema: (`id`, `score`, `j`). `j` is the attribute with respect to which the relations are joined (i.e., `R.j` = `S.j` = `T.j` in a join result) and the results are ranked by `sum`{`R.score`, `S.score`, `T.score`}. The selectivity of the join is 0.2% (we also experimented with different join selectivities and derived similar results). The three relations are ranked by `score` and their tuples are retrieved incrementally. For HRJN, we used the evaluation plan (`R` ⋈ `S`) ⋈ `T` (other plans have similar performance). Figure 8c plots the number of tuples accessed by LARA-J and HRJN until they output the same number of results. This reflects the *output rate* of the approaches (i.e., how many results they can produce after a specific number of accesses). Observe that LARA-J produces results much earlier than HRJN. The space used by the two methods to accommodate intermediate results (not shown in the graph) is roughly proportional to the number of accesses. Both methods are computationally efficient (200 results are output in just 78 msec by LARA-J and 94 msec by the plan of HRJN operators). HRJN's efficiency is due to the computationally cheap threshold bound it uses; however, more accesses are required to compute the same result as LARA-J. This experiment not only demonstrates the applicability of LARA to top-k join queries but also indicates that multiway top-k operators can be more effective (in terms of accesses) than trees of binary join operators.

[4] We used a multiway implementation of NRA-RJ, instead of a tree of binary NRA-RJ operators. A binary operator tree for $m > 2$ incurs many more object accesses, as discussed in [9].



(a) exact scores (UI, $m = 3, k = 20$) (b) incremental top-k (UI, $n = 50$K, $m = 3$) (c) top-k joins, (UI, $n = 50$K, $m = 3$)

Figure 8. Variants of top-k search ($\gamma = \texttt{sum}$)

attributes of FC	NRA	LARA	LARA-OPT
{*hh, hr, hf*}	56582	56582	54356
{*hh, vh, hr*}	100579	100579	100579
{*hh, vh, hf*}	113663	113663	86023
{*hh, hr, hf, vh*}	240196	240196	210315

(a) number of accesses

attributes of FC	NRA	LARA	LARA-OPT
{*hh, hr, hf*}	112 (92)	0.22 (0.16)	0.22 (0.16)
{*hh, vh, hr*}	189 (152)	0.36 (0.29)	0.36 (0.29)
{*hh, vh, hf*}	211 (179)	0.47 (32)	0.39 (0.3)
{*hh, hr, hf, vh*}	654 (412)	1.1 (0.72)	1.1 (0.72)

(b) time in seconds

Figure 9. Forest coverage ($\gamma = \texttt{min}$, $k = 20$)

Finally, we compare LARA with NRA for top-k min queries. For the NRA implementation, we used the optimization discussed in Section 5.4.1, where sources with $l_i \leq t$ are immediately pruned. We implemented two versions of LARA; one that uses that optimization and another (LARA-OPT) that expands the input with the largest l_i. Figure 9 compares the three algorithms on the FC dataset. Note that LARA-OPT may incur fewer object accesses compared to LARA and NRA, due to the more optimized scan of the inputs. On the other hand, LARA-OPT is not always faster than LARA, because it has the additional complexity of maintaining the input with the largest l_i.

7 Conclusions

In this paper we proposed a new algorithm for processing top-k queries by sequentially accessing sources of ranked atomic object scores. LARA is based on some core observations about the behavior of all "no-random-accesses" (NRA) algorithms. The main advantage of LARA compared to previous NRA implementations is its high efficiency at no cost of redundant object accesses. LARA employs a lattice to facilitate efficient computation of the result and easy detection and pruning of sources that do not contribute to the result. Experimental comparison with previous NRA implementations, show that LARA is orders of magnitude faster. In addition, LARA incurs fewer object accesses; the savings are marginal for synthetic data, but can be significant for real data. Finally, LARA has minimal memory requirements, since no unseen objects in the growing phase are considered in the shrinking phase. We have also shown how LARA can be adapted for several top-k variants (exact scores, on-line and incremental search, top-k joins, other aggregate functions, weighted search, etc.) In the future we plan to optimize LARA for top-k variants, especially for complex aggregate functions and top-k joins.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[3] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD Conference*, 2002.

[4] R. Fagin. Combining fuzzy information from multiple systems. *J. Computer System Sci.*, 58(1):83–99, 1999.

[5] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, 2003.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[7] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries in image databases. In *VLDB Conference*, 2000.

[8] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *IEEE Int'l Conf. on Information Technology (ITCC)*, 2001.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB Conference*, 2002.

[10] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB Conference*, 2003.

[11] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD Conference*, 2004.

[12] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB Conference*, 2001.

[13] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.

[14] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB Conference*, 2004.