Efficient Top-k Aggregation of Ranked Inputs

NIKOS MAMOULIS
University of Hong Kong
MAN LUNG YIU
Aalborg University
KIT HUNG CHENG
University of Hong Kong
and
DAVID W. CHEUNG
University of Hong Kong

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 with monotone aggregate functions. We provide an analysis for its cost and show that it is always no worse than the baseline "no random accesses" algorithm in terms of computations, accesses, and memory required. As a side contribution, we perform a space analysis, which indicates the memory requirements of top-k algorithms that only perform sorted accesses. For the case, where the required space exceeds the available memory, we propose disk-based variants of our algorithm. We propose and optimize a multiway top-k join operator, with certain advantages over evaluation trees of binary top-k join operators. Finally, we define and study the computation of top-k cubes and the implementation of roll-up and drill-down operations in such cubes. Extensive experiments with synthetic and real data show that, compared to previous techniques, our method accesses fewer objects, while being orders of magnitude faster.

Categories and Subject Descriptors: H.2 [**Database Management**]: General; H.3.3 [**Information Search and Retrieval**]: General

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Top-k queries, rank aggregation

This work was supported by grant HKU 7160/05E from Hong Kong RGC. Nikos Mamoulis, Kit Hung Cheng, and David W. Cheung are with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. Their emails are (in author list order) nikos@cs.hku.hk, khcheng9@graduate.hku.hk, and dcheung@cs.hku.hk. Man Lung Yiu is with the Department of Computer Science, Aalborg University, DK-9220 Aalborg, Denmark. His email is mly@cs.aau.dk. A preliminary version of this paper appears in [Mamoulis et al. 2006].

1. INTRODUCTION

Several applications combine ordered scores of the same set of objects from different (potentially 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. Additional applications include the retrieval of images according to their aggregate similarity to an example image with respect to various features, like color, texture, shape, etc. [Ortega et al. 1997] and the e-commerce services sorting their products according to user preferences to facilitate purchase decision [Agrawal and Wimmers 2000].

This problem, known as the top-k query, has received considerable attention from the database and information retrieval communities [Ortega et al. 1997; Chang et al. 2000; Fagin et al. 2001; Fagin 2002; Kießling 2002; Ilyas et al. 2004]. Fagin's early algorithm [Fagin 1999], later optimized in [Nepal and Ramakrishna 1999; Güntzer et al. 2000; Fagin et al. 2001], 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 [Güntzer et al. 2001; Natsev et al. 2001; Ilyas et al. 2002; 2003] for several reasons. First, in many applications, random accesses to scores are impossible [Fagin et al. 2001]. 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 [Ilyas et al. 2003], produced incrementally and/or on-demand from remote services or underlying database operators, where individual scores of random objects are not available at anytime.

[Fagin et al. 2001] 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 [Güntzer et al. 2001; Natsev et al. 2001; Ilyas et al. 2002; 2003], in practice 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., [Nepal and Ramakrishna 1999]), have large buffer requirements [Natsev et al. 2001; Ilyas et al. 2002].

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 that perform top-k search with monotone aggregate functions. 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, 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. The proposed $\underline{\text{La}}$ttice-based $\underline{\text{R}}$ank $\underline{\text{A}}$ggregation (LARA) algorithm, during the shrinking phase, employs a lattice to minimize its computational cost. LARA can be implemented as a standalone rank aggregation tool or as a multiway merge join operator for dynamically produced ranked inputs. We analyze the time and space complexity of our method and demonstrate its superiority to previous baseline implementations of NRA top-k retrieval; its per-access computational cost is only $O(\log k + 2^m)$, where m is the number of inputs that are merged. For the case, where the space requirements of our method exceed the available memory, we propose extensions that perform disk-based management of the candidate top-k objects.

As a third contribution, we discuss how our algorithm can be seamlessly adapted for top-k search variants. We present an extension of our algorithm that can be used as a top-k join operator [Natsev et al. 2001; Ilyas et al. 2003], suitable for queries that request the results of a (multiway) join to be output in order of some aggregate score. The technique we propose "pushes" the ordering predicate in the ripple-join-like evaluation component. We propose an optimization of our top-k join algorithm for sparse join graphs, where the maintenance of partially joined tuples that correspond to Cartesian products is avoided. In addition, we propose an interesting variant of top-k search in an OLAP context; given a set of m ranked inputs and an aggregate function γ, retrieve for each of the $2^m - 1$ combinations of inputs the top-k objects by applying γ only to them. We show how LARA can be adapted to compute top-k cubes efficiently. Finally, we define *browsing* (roll-up and drill-down) operations among top-k results of different cuboids. Extensions of LARA that perform browsing *incrementally* (i.e., by continuing search from the state where the result of the previous query was finalized) are proposed.

We conduct an extensive experimental evaluation with synthetic and real data and show that, compared to previous techniques, our method accesses (sometimes significantly) fewer objects, while being orders of magnitude faster. The experiments also demonstrate the efficiency and practicality of LARA extensions on top-k variants, as well as the accuracy of our theoretical analysis.

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. A time/space complexity analysis for LARA and NRA algorithms in general is presented in Section 5 together with several optimizations that improve the efficiency of our algorithm in practice. In Section 6, we discuss 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 7. Finally, Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

This section formally defines top-k queries and provides a literature review for the basic problem and its variants, discusses other related problems and sets the focus of this paper.

2.1 Problem definition

Let D be a collection of n objects (e.g., images) and $S_1, S_2, \dots, S_m$ 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 $x_1, x_2, \dots, x_m$ of an object x in $S_1, S_2, \dots, S_m$ 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 $S_1, S_2, \dots, S_m$ (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 $\gamma_b = \gamma(0.6, 0.8, 0.8) = 2.2$.

2.2 Fagin's algorithms

[Fagin et al. 2001] 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 [Fagin et al. 2001; Nepal and Ramakrishna 1999; Güntzer et al. 2000]) retrieves objects from the ranked inputs in a round-robin fashion[1] 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; $T = \gamma(l_1, \dots, l_m)$ 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.

Consider again the example of Figure 1 and assume that the aggregate function is $\gamma = $ `sum` and $k = 1$. In the first round, TA retrieves objects c (from S_1 and S_3) and a (from S_2). Since c has not been seen before and its aggregate score is incomplete, a random access in performed to S_2 to access c's score there and derive $\gamma_c = 2.0$.

[1]In fact, the access of the ranked inputs needs not be round-robin by essence; we discuss this access pattern for the ease of discussion. All presented algorithms can operate independently of the order by which information is accessed.

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

Fig. 1. Three ranked inputs

Similarly, two random accesses are performed to compute $\gamma_a = 1.9$. After the first round, c is the best object found so far, but γ_c is lower than $T = \gamma(0.9, 0.9, 0.9) = 2.7$. Thus, it is likely that a better solution can be found and the algorithm proceeds to the next round. There, objects d,b, and a are accessed, the aggregate scores of d and b are computed by random accesses (a has been seen before, so it is ignored), and b is found to be the top object (with $\gamma_b = 2.2$). A better object can still be found ($T = 2.5$), so the algorithm proceeds to the next round, retrieving the new object e (but still b remains the best object seen so far). A fourth round is not required since now $T = 2.0 \leq \gamma_b$. Thus, TA terminates returning b, after 9 sorted and 9 random accesses.

For the case where random accesses are either impossible or much more expensive compared to sorted ones, [Fagin et al. 2001] 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 and 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 $\gamma_x^{lb}$;
3. if less than $k$ objects have been seen so far then goto Line 1;
4. for each object $x$ seen so far compute $\gamma_x^{ub}$;
5. $W_k$ := the $k$ objects with the highest $\gamma^{lb}$;
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;
```

Fig. 2. The NRA algorithm

Let us see how NRA processes the top-1 query for the ranked inputs of Figure 1 and $\gamma = \texttt{sum}$, 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.

2.3 Variants of NRA

A simple variation of the basic NRA algorithm is Stream-Combine (SC) [Güntzer et al. 2001]. 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.

[Ilyas et al. 2002] implemented NRA as a "partially" non-blocking operator, 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). The idea is to maintain the threshold of TA (i.e., the aggregate of the last seen atomic scores at all sources) and report objects as soon as their worst-case score is larger than the threshold and the best-case scores of all partially seen objects.

2.4 Top-k joins

The top-k query we have seen so far is a special case of top-k *join* queries [Natsev et al. 2001; Ilyas et al. 2003; Ilyas et al. 2004], 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 `id`s are unique, all `R`, `S`, `T` have the same collection of tuple `id`s, and tuples from each relation are ranked based on their scores. [Natsev et al. 2001; Ilyas et al. 2003] propose algorithms for solving generic top-k joins. The J* algorithm [Natsev et al. 2001], for each input stream, defines a variable whose domain is the values from that stream. The goal is to find a valid assignment (based on the join conditions) for all variables of maximal aggregate score. Each partial join result (i.e., valid assignment for a subset of variables) is called a *state* and has an upper bound for the aggregate scores of the complete join results that include it. The algorithm maintains a heap for all partial and complete states (i.e., complete join results). At each step, the state at the top of the heap is popped, missing values are sought for it (if partial) by accessing the corresponding streams and the results are pushed back on the heap. The algorithm terminates if the top state in the heap is a complete one. J* can produce ranked join results *incrementally.*

[Ilyas et al. 2003] 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. Let h_i and l_i be the highest and lowest scores seen so far in S_i. The threshold used by [Ilyas et al. 2003] is $\max_{i=1}^{m} \gamma(h_1, \ldots, h_{i-1}, l_i, h_{i+1}, \ldots, h_m)$. They also focus on the implementation of a binary top-k join operator, whose instances can be combined in evaluation trees for multiway top-k queries. This operator, called hash-based rank-join (HRJN), is based on iterator functions that read the tuples of each of the two ranked inputs and probe them against the tuples of the other input that have already been seen. Join results are organized in a priority queue. Join results in the queue having aggregate score larger than the threshold $T = \max\{\gamma(h_{left}, l_{right}), \gamma(l_{left}, h_{right})\}$ are guaranteed to have higher aggregate score than any join result not produced yet, so they are incrementally output. Here, h_{left}, l_{left} (h_{right}, l_{right}) denote the highest and lowest atomic score values seen in the left (right) input. In this paper, we propose an efficient non-binary operator for top-k queries which can also be adapted for generic rank joins, as we show in Section 6.1.

2.5 Other related work

The methods we discussed so far in Sections 2.2 and 2.3 focus on top-k query processing for middleware, where the input sources are distributed. [Marian et al. 2004] study top-k queries for web data 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. [Chang and Hwang 2002] study top-k query evaluation for the case where random accesses are only possible for some query components, due to the involvement of expensive predicates.

Top-k queries have also been studied for centralized, relational databases. [Bruno et al. 2002] process top-k queries, after converting them to multidimensional range queries. They utilize multidimensional histograms to accelerate search. Other work on expressing and evaluating top-k and other preference queries in relational databases includes [Carey and Kossmann 1997; Agrawal and Wimmers 2000; Kießling 2002].

Materialization and maintenance of top-k query results has been studied in [Hristidis and Papakonstantinou 2004] and [Yi et al. 2003]. A recent work on the maintenance of top-k query results for streaming data constrained by sliding windows is [Mouratidis et al. 2006]. Index-based approaches for computing top-k query results for centralized data are presented in [Tsaparas et al. 2003; Tao et al. 2007]. Finally, probabilistic extensions of top-k algorithms for approximate retrieval (based on underlying indexes) have been proposed in [Theobald et al. 2004].

There are several other problems which are highly related to top-k queries. In particular, the popular nearest neighbor problem (a.k.a. similarity search) in intervalscaled high dimensional data can be seen as a special case of top-k search, where the atomic scores of objects map to absolute attribute value differences to a reference object and there is a preference function (usually an L_p measure) that combines them. Thus, certain types of top-k queries can be evaluated by techniques for similarity search in multidimensional data [Roussopoulos et al. 1995; Beyer et al. 1999; de Vries et al. 2002]. [Balke and Güntzer 2004] extends the concept of top-k queries to more generic multi-objective queries that also include the popular skyline query [Börzsönyi et al. 2001].

The focus of this paper is on the implementation of efficient top-k aggregation of ranked inputs, without relying on indexes or precomputed materialized views. There are several reasons why index-based approaches or materialized views may be inapplicable or infeasible in practice. First, the combination of possible aggregation attributes and merging functions can be very large and it might be impractical to create and maintain indexes or views for all these combinations. If the number of attributes that might be considered for aggregation is m, there are $2^m - 1 - m$ combinations of two or more attributes to be considered for indexing or top-k view materialization. This number has to be multiplied with the number of functions that could be considered for aggregation (e.g., weighted `sum`, `min`, hybrid functions, etc.). Besides, the space occupied by these views and indexes might be too large. In addition, such approaches may not applicable for the case where the ranked lists are produced from distributed (e.g., web) sources, due to unavailability of on-line data or privacy constraints.

Like most of the past research on top-k aggregation of sorted lists [Güntzer et al. 2001; Natsev et al. 2001; Ilyas et al. 2002; 2003], we focus on the case where only sequential accesses are allowed (i.e., NRA top-k search). Random accesses may be impossible or very expensive if the lists are generated by distributed servers (e.g., web services). Even in centralized databases, the difference between random and sequential I/Os increases over the years, due to the mechanical operations involved in random disk seeks. Finally, we may want to merge (possibly unbounded) streams of ranked inputs [Ilyas et al. 2003], produced incrementally and/or on-demand from remote services or underlying database operators, where individual scores of random objects are not available at anytime. Since NRA [Fagin et al. 2001] has already been shown optimal in terms of number of accesses (although there can be significant practical differences between implementations of the algorithm [Ilyas et al. 2002]), our goal is to optimize the computational performance of this method, subject to keeping the access cost minimal. In the next section, we present some observations that can help toward achieving this goal.

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. Provided that k is a priori known, these observations impose two phases that any NRA algorithm essentially goes 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 lemmas that impose some useful rules toward 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:

LEMMA 1. If $t < T$, every object which has not been seen so far at 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. □

LEMMA 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 lemma 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. □

Lemmas 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:

Lemma 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 Lemma 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, Lemmas 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 Lemma 3) helps reducing the accesses during the shrinking phase.

Corollary 1. If $t \geq T$ and all current candidate objects have already been seen at 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 La̲ttice-based R̲ank A̲ggregation (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[2] 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

[2]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}.

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 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 Lemma 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 [Fagin et al. 2001; Güntzer et al. 2001; Ilyas et al. 2002]) 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[3] 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 $\gamma^{ub}_{x^v}$ 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^{lb}_x 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^{prev}_x where x belonged, before it was accessed at S_i. If so, a new leader for v^{prev}_x 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^{prev}_x 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

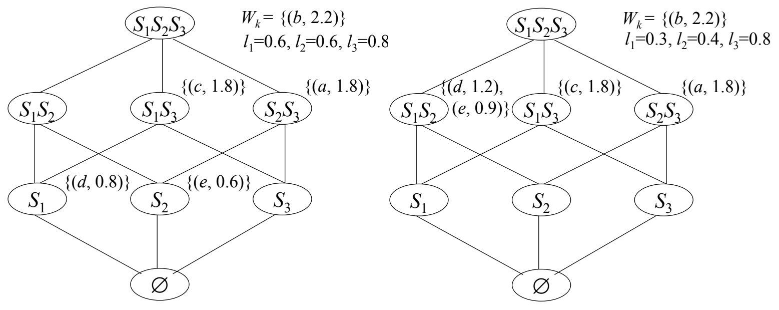
LARA, as presented so far, is described by the pseudocode of Figure 3. The algorithm repeatedly accesses objects from the various inputs and depending on whether it is in the growing or shrinking phase it performs the appropriate operations. As an example of LARA's functionality, consider again the top-1 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 $t = \gamma^{lb}_b = 2.2$ 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 $\gamma^{ub}_{x^{S_1S_3}} = \gamma^{ub}_c$, LARA adds l_2 (the highest possible score c can have in S_2) to γ^{lb}_c. Since $\gamma^{ub}_{x^{S_1S_3}} > t$, LARA proceeds to access the next object from S_1, which is e. Now, γ^{lb}_e becomes $0.9 < t$ and the object is promoted to node S_1S_2. We still have $\gamma^{ub}_{x^{S_1S_3}} > t$, thus LARA accesses the next object from S_2, which is d. Now, γ^{lb}_d becomes $1.2 < t$ and the object is promoted to S_1S_2. Figure 4b shows the lattice at this stage. Note that now $\gamma^{ub}_{x^v}$ for every (occupied) lattice node is at most t (i.e., $\gamma^{ub}_{x^{S_1S_2}} = \gamma^{ub}_d = 2.0, \gamma^{ub}_{x^{S_1S_3}} = \gamma^{ub}_c = 2.2, \gamma^{ub}_{x^{S_2S_3}} = \gamma^{ub}_a = 2.1$), 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 (useless) objects never been seen during

[3]Top-k queries usually combine a small $m \leq 10$ number of ranked inputs [Fagin et al. 2003]. Thus, in typical applications, $n \gg 2^m$. In Section 5.1 we analyze the per-access cost of LARA and compare it to that of simple NRA.

```
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 $\gamma_x^{lb}$; /* partial aggregate 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 $\gamma_x^{lb}$; /* partial aggregate 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;
```

Fig. 3. The LARA algorithm



(a) after 9 accesses (b) after 11 accesses

Fig. 4. The lattice at two stages of LARA

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.

5. ANALYSIS AND OPTIMIZATIONS

In this section, we analyze the time and space complexity of LARA and compare it to that of NRA. Throughout our analysis, we assume that the rankings of objects at different inputs are statistically independent and that we follow a round-robin access schedule. Although we consider the aggregate function to be `sum` in the analysis, our results can be easily extended to other common monotone functions (e.g., `min`). After analyzing the expected complexity of LARA (based on the above assumptions), we propose some optimizations that may reduce the computational cost and the number of accesses by LARA, in practice. In addition, we propose adaptations of LARA for the case where the volume of candidates to be managed exceeds the capacity of the main memory.

5.1 Time complexity

As discussed in Section 4.1, the per-access computational cost of LARA in the growing phase is $O(\log k)$. Now assume that we are in the shrinking phase and we access an atomic score from source S_i. Assume also that we have accessed so far $\alpha \cdot n$ atomic scores from each source ($0 < \alpha < 1$). In other words, α denotes the probability that a random object has been seen at a specific source. We have the following possibilities:

—*The accessed atomic score belongs to an object x that has never been seen before* ($x \notin H$). The probability for this to happen is $(1-\alpha)^m$. The object is just pruned in this case and the cost of LARA to update the lattice is just $O(1)$.

—*The atomic score belongs to an object x that has been seen before.* The probability for a random object y, seen at some but not all sources, to currently belong to a node v of the lattice $\mathcal{L}$ with arity l is $P_{y \in v} = \frac{\alpha^l (1-\alpha)^{m-l}}{(1-(1-\alpha)^m)-\alpha^m}$. The nominator corresponds to the probability that y has been seen at a particular combination of l sources and the denominator is the probability that the object has been seen in at least one $(1-(1-\alpha)^m)$ but not at all sources (α^m). In other words, the current cardinality of a node $v \in \mathcal{L}$ is $P_{y \in v} \cdot |C|$, where $|C|$ is the number of candidate objects (excluding those seen at all sources). Given a particular node v, and because we assume that the rankings of objects in different sources are independent, the probability of v's leader to be the currently accessed object x is $\frac{1}{P_{y \in v} \cdot |C|}$, since $P_{y \in v} \cdot |C|$ is the expected number of objects in v. Thus, the probability of x currently being a leader in one of the nodes of the lattice $\mathcal{L}$ is $\sum_{v \in \mathcal{L}, S_i \notin v} \frac{1}{P_{x \in v} \cdot |C|} P_{x \in v} = \frac{(2^m-2)/2}{|C|}$.
When x is accessed at S_i, LARA (i) potentially updates W_k, (ii) potentially updates the leader of the lattice node v_x^{prev} where x existed, and (iii) potentially updates 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 $O(|C|)$, where $|C|$ is the number of candidates, since we need to scan the entire candidates set C 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 probability $\frac{(2^m-2)/2}{|C|}$, as shown above. Thus, the expected per-access cost of LARA due to (ii) is $\frac{(2^m-2)/2}{|C|} \cdot O(|C|) = O(2^m)$.[4] The cost of operation (iii) is $O(1)$, since a mere comparison to the previous leader is required. Summing up, for each access in the shrinking phase, the expected

[4]In our experiments, we verified that leader promotions in typical runs of LARA are very few (less than ten).

S_1	S_2	S_3
o_1 0.9	o_{34} 0.9	o_{67} 0.9
o_2 0.9	o_{35} 0.9	o_{68} 0.9
...	...	...
o_{33} 0.9	o_{66} 0.9	o_{99} 0.9
o_{100} 0.4	o_{100} 0.4	o_{100} 0.4
o_{67} 0.25	o_1 0.25	o_{34} 0.25
o_{68} 0.25	o_2 0.25	o_{35} 0.25
...	...	...
o_{99} 0.25	o_{33} 0.25	o_{66} 0.25
o_{34} 0.1	o_{67} 0.1	o_1 0.1
o_{35} 0.1	o_{68} 0.1	o_2 0.1
...	...	...
o_{66} 0.1	o_{99} 0.1	o_{33} 0.1

Fig. 5. A worst-case scenario for LARA

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. The worst-case scenario for LARA is that, during the shrinking phase, every access causes a promotion of a leader. A (pathological) example for $n = 100$, $m = 3$, $k = 1$, and $\gamma = \texttt{sum}$ is shown in Figure 5. LARA will enter the shrinking phase when o_{100} is accessed. Then $t = 1.2$, whereas $\gamma^{ub}_{x^{S_1}} = \gamma^{ub}_{x^{S_2}} = \gamma^{ub}_{x^{S_3}} = 1.7$. From this point and until o_{34} is accessed at S_1, LARA promotes the current leader of a singleton node (i.e., S_1, S_2, or S_3) at a doubleton node (S_1S_2, S_2S_3, or S_1S_3, respectively). LARA will terminate after another leader promotion (when o_{34} is accessed at S_1) and two more accesses (when o_1 is accessed at S_3). As a result, for almost all its accesses in the shrinking phase (i.e., half the overall accesses) the computational cost of LARA is O($|C|$). Even in such a rare, worst-case scenario, LARA maintains its significant advantage over NRA in the growing phase. In order to minimize the effect of such pathological cases and at the same time achieve better scalability with m, in Section 5.3.1 we suggest an alternative implementation of LARA (called LARA-MAT).

The time complexity of NRA is dominated by the computation and maintenance of the upper score bounds of *all* objects that have been partially seen. In other words, the cost of NRA in both phases is O($|C|$) per access. We now analyze the relationship between $|C|$ and n, as a function of the fraction α of atomic scores accessed at each source. Assume that we have accessed $\alpha \cdot n$ atomic scores from each source ($0 < \alpha < 1$). Now suppose that we access an atomic score from source S_i. Our objective is to estimate the number of objects for which we have to update their upper bound scores after this access. This corresponds to the number of objects seen at any source, but not S_i. The probability of a random object x not seen at source S_i is $P_{\neg S_i} = (1-\alpha)$. The probability of the same object x seen at one or more sources other than S_i is $P_{any\ S_j | S_j \neq S_i} = (1 - (1-\alpha)^{m-1})$. Therefore, the expected number of objects for which NRA has to update their upper bound after an access is $P_{\neg S_i} \cdot P_{any\ S_j | S_j \neq S_i} \cdot n$ or $(1 - (1-\alpha)^{m-1})(1-\alpha)n$. As we show in the next paragraph (space analysis), NRA requires accessing large fractions of the lists before it can terminate, therefore the O($|C|$) per-access complexity of NRA grows much higher than the O(2^m) per-access cost of LARA, as the number of candidates $|C|$ increases during the growing phase.

5.2 Space complexity

The maximum space required by LARA corresponds to the maximum number of candidates that need to be maintained. Based on the lemmata of Section 3, this corresponds to the number of distinct objects accessed until the shrinking phase begins (i.e., until $t \geq T$). Our space complexity analysis is based on the assumption that the distribution of scores in the sources are uniform and independent. This assumption implies that the atomic score of the $(\alpha \cdot n)$-th top object at any list S_i is $1 - \alpha$. Assume that we are still in the growing phase and $\alpha \cdot n$ objects have been accessed from each source. Then, $T = m \cdot (1 - \alpha)$, assuming $\gamma =$ `sum`. At this stage, the expected number of objects which have been seen at all m sources is $n(\alpha)^m$. In general, at exactly m' sources, we expect to have seen $n\binom{m}{m'}(\alpha)^{m'}(1-\alpha)^{m-m'}$ objects. In addition, the probability of an object seen at all sources to have aggregate score at least T is 1, whereas the expected γ_x^{lb} bound of an object x seen exactly at m' sources is $m'(1 - \frac{\alpha}{2})$. Overall, assuming that we have accessed $\alpha \cdot n$ scores from each source, the expected number of objects for which the lower bound is at least T is:

$$\sum_{m'=1}^{m} \left(\lfloor n\binom{m}{m'}(\alpha)^{m'}(1-\alpha)^{m-m'} \rfloor \times (m'(1-\frac{\alpha}{2}) \geq T) \right) \qquad (1)$$

The second factor of each summed term is a boolean formula that translates to integer 1 or 0. The goal is to find the smallest α for which the sum above is at least k. This can be achieved by numerical analysis (i.e., using the bisection method). Then, based on the found α, the expected distinct number of candidates that enter the shrinking phase (i.e., the space complexity of LARA) is $n(1-(1-\alpha)^m)$. NRA can also achieve the same space complexity, if any newly seen object whose upper bound is at most t is not stored (this happens as soon as $t \geq T$).

Figure 6a visualizes our analysis for $m = 2$ and γ=`sum`. If $\alpha \cdot n$ atomic scores have been seen from each source, the space can be divided into three regions; (i) *completely* seen region (dark gray), whose objects have been accessed from all sources, (ii) *partially* seen region (light gray), whose objects have been accessed from some but not all sources, and (iii) *unseen* region (white), whose objects have not been accessed from any source. Under our uniformity assumption, the number of objects in each region is estimated as the area of the region multiplied by n. The candidate size corresponds to the total number of objects in dark gray and light gray regions. In Figure 6b, the dotted contour line represents all objects having aggregate score t (i.e., the current lowest score in W_k). The aggregate score of any object lying below the contour is less than t. The growing phase ends when the contour does not intersect the unseen region. In the shrinking phase, among the partially seen objects, those having upper bound scores smaller than t can be pruned as they cannot lead to better result. In Figure 6b, these objects reside in the bold-framed rectangles. As LARA accesses more scores, α increases and the non-pruned partially seen regions are trimmed. Eventually, the number of candidates drops to zero and LARA terminates.

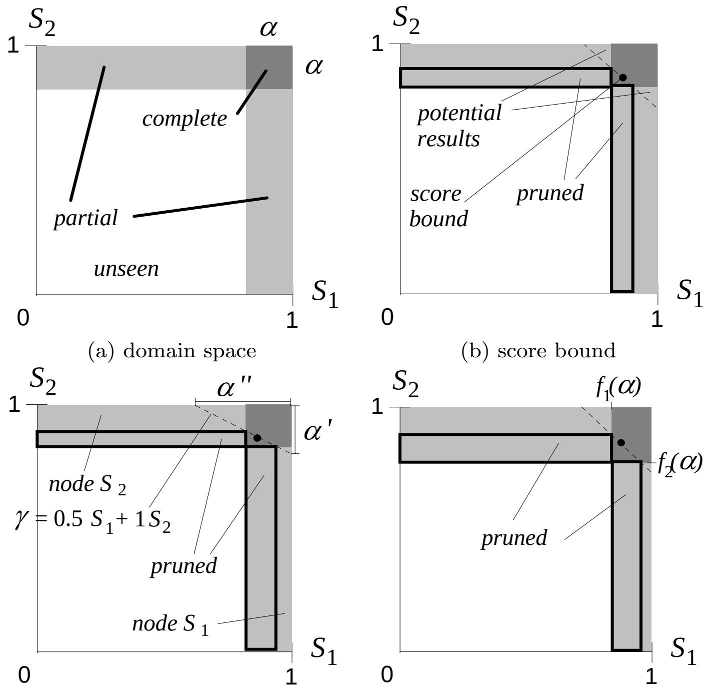


Fig. 6. Geometric analysis

Although our space analysis assumes uniform distribution of atomic scores, it can be easily extended for the case of non-uniform (i.e., real) datasets, provided that (i) the sources are independent, and (ii) for each source S_i, a histogram for the scores distribution there is available. In this case, Equation 1 is replaced by:

$$\sum_{\text{combination } v} \left(\lfloor n \cdot (\alpha)^{arity(v)} \cdot (1-\alpha)^{m-arity(v)} \rfloor \times ((\sum_{i \in v} \overline{S_i[f_i(\alpha),1]}) \geq T) \right) \quad (2)$$

In the above equation, for every combination v of sources (i.e., every node of the lattice) we estimate the number of objects seen exactly at these inputs (first factor in sum). In the second factor of each summed term, $\sum_{i \in v} \overline{S_i[f_i(\alpha),1]}$ is the expected γ_x^{lb} of an object x seen at combination v. In this quantity, $f_i(\alpha)$ is the $(\alpha \cdot n)$-th atomic score at input S_i and $\overline{S_i[a,b]}$ is the mean score of the items in S_i whose score is in the interval $[a,b]$. Both $f_i(\alpha)$ and $\overline{S_i[a,b]}$ can be easily estimated from the histogram of S_i.

If multi-dimensional histograms are available, which not only capture the score distributions at the different input, but also the correlation between them, then we can derive an even more accurate formula. We can estimate the objects seen at a combination v by a multi-dimensional range-count query with extent $[f_i(\alpha),1]$ at the inputs (i.e., dimensions) that appear in v and $[0, f_i(\alpha)]$ at the inputs not in v. From these objects, we can estimate the ones with γ_x^{lb} at least T again by a range-count query with constraint $\gamma_x \geq T$, where γ applies only to the dimensions in v. To illustrate this method, consider the geometric example of Figure 6a and assume that a 2D histogram is available. We can estimate the number of objects seen only at combination $v = \{S_2\}$, after accessing $\alpha \cdot n$ objects from both inputs, by applying a range-count query having as extent the upper light gray rectangle. Assuming that the dotted contour line in Figure 6b represents the objects having aggregate score T, we can estimate the number of objects $x \in v$ with γ_x^{lb} at least T by a range-count query having as extent the upper light gray rectangle excluding its bold-framed part. If we repeat this operation for all combinations v, we can derive an estimate for the total number of candidates for any value of T, which allows us to apply a numerical analysis technique for estimating the space requirements of LARA.

5.3 Optimizing the performance of LARA in practice

In this section, we present several optimizations of LARA that minimize its number of accesses and the computational cost in practice. In a nutshell, we (i) prune candidates as soon as they are known to have lower upper bound than t, (ii) use Corollary 1 to detect and dry up inputs that cannot contribute to the top-k result, and (iii) delay the expensive computation and update of upper bounds when entering the shrinking phase.

5.3.1 *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.

An implementation of LARA that efficiently performs the second optimization and also scales better with m *materializes* the lattice; i.e., candidates are explicitly partitioned into sets according to the lattice nodes where they belong. In this LARA-MAT implementation, leader updates do not require the scanning of the whole candidate set, but only the objects currently in the node where the leader should be updated. The price to pay is that at each access (e.g., of object x), the contents of two nodes must be updated (e.g., x is deleted from v_x^{prev} and inserted to v_x). Therefore, if the leader of a node v has γ^{ub} not greater than t, we can immediately prune the objects in v, without having to access all candidates. In addition, if a leader is promoted from node v, the new leader for v can be selected from the objects that belong to v only (i.e., there is no need to scan the whole set C of candidates). If m is large (i.e., the lattice is large), the probability that the currently accessed object is a leader in a node increases, thus leader promotions become frequent. In this case, LARA-MAT is expected to be more efficient than LARA. Finally, for large values of m there can be many nodes of the lattice that are empty. Thus, the comparison of t with the upper bounds of leaders (see Line 24 of Figure 3) is restricted to checking only the non-pruned leaders, which are much fewer than 2^m. As a result, the per-access cost of LARA-MAT can be much smaller than that of the original algorithm which performs at least 2^m comparisons per access.

5.3.2 *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) $\gamma^{ub}_{x^v} \leq t$. 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 $\gamma^{ub}_x > t$. We can "dry up" inputs by exploiting Corollary 1 as follows. Let $v = S_1S_2 \dots S_{i-1}S_{i+1} \dots S_m$ 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, it never performs more accesses. Thus, LARA is *instance optimal* [Fagin et al. 2001] 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.

The impact of drying up inputs to the access cost of LARA can be geometrically demonstrated in the example of Figure 6c. Consider the weighted sum aggregate function $\gamma = 0.5 \cdot S_1 + 1 \cdot S_2$. Recall that each partially seen region (in light gray) corresponds a lattice node in LARA. In the figure, the topmost (rightmost) region corresponds to the objects seen only at node S_2 (S_1). Since the (score bound) contour is slanted towards the S_2 axis, the number of candidates at lattice node S_1 is much fewer than that those at node S_2. As LARA continues accessing the sources, at some stage the width of the horizontal strip becomes α' (i.e., $\alpha' \cdot n$

scores are accessed from S_2). At that time, node S_1 becomes empty (all candidates there are pruned since the contour is above the a' point). Thus, LARA dries up source S_2 and accesses S_1 until the width of the vertical strip grows to α''. On the other hand, NRA continues expansion along both sources until the width of both strips become α''. Therefore, the access cost saving of LARA over NRA can be significant (especially for functions with vastly different weights). Drying up sources can also save many accesses at uniform functions (like `sum`) on non-uniform (e.g., real) datasets. In this case, when the same number of scores have been accessed from each source, different strips have different widths for non-uniform sources. For instance, in Figure 6d, the $\alpha \cdot n$-th score at S_1 (i.e., $f_1(\alpha)$) is larger than the $\alpha \cdot n$-th score at S_2 (i.e., $f_2(\alpha)$), thus the vertical light gray strip is thinner than the horizontal light gray strip. This causes the contour to prune the candidates in S_1 and to dry S_2 up, early.

5.3.3 *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 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 $u \leq t$, 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 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). A subtle thing to note for this optimization is that together with the initial computation of u we also keep track of the leader which is responsible for u. If this leader enters W_k, the actual upper bound may drop by a value larger than Δl; in that case, the actual upper bound u and the corresponding leader are re-computed.

A closer look reveals that the above computational optimization may weaken the power of the access cost optimization of Section 5.3.2. Even when the highest upper bound of lattice leaders is greater than t, we can dry up a source S_i provided that (i) all objects in W_k have already been seen at S_i and (ii) for each lattice node v that does not include S_i, $\gamma_{x^v}^{ub} \leq t$ holds. In order to preserve the full power of the access cost optimization, we can modify the above computational optimization as follows. Instead of using the maximum upper bound score of the lattice, we take $u = min\{u_1, u_2, ..., u_m\}$, where u_i is the maximum upper bound score of all nodes without the source S_i. We decrement u by Δl each time a source is accessed, until $u - \Delta l \leq t$. At that time, we recompute u and also check whether any source can be dried up.

5.4 Disk-based management of candidates

Existing "no random accesses" algorithms [Fagin et al. 2001; Natsev et al. 2001; Ilyas et al. 2002] assume that the memory is large enough to accommodate all candidates, which may not hold in practice. The space analysis of Section 5.2 and our experimental evaluation unveils that the number of top-k candidates during LARA (or NRA) could grow to a large percentage of the total number of objects. In problem settings, where the system's memory is too small to fit the partial score of every candidate, the application of LARA (and NRA algorithms in general) is infeasible.

In this section, we propose adaptations of LARA for bounded memory, which manage candidates and their partial scores on disk. Our objective is to minimize the disk I/O, while keeping the number of accesses close to the minimum possible: the accesses incurred by a version of LARA which has unlimited memory. We emphasize that the number of accesses accounts for the middleware cost and its interpretation is different from the I/O cost for managing candidates on disk.

Figure 7 shows the pseudo-code of an *eager* algorithm for top-k processing with a memory constraint B. Let B be the maximum number of entries that the hash table H can accommodate. During the growing phase, the algorithm operates exactly like LARA until H becomes full. At this stage, the candidates in H are sorted by object id, written to a disk file $\mathcal{Z}$, and H is cleaned up. LARA-EAGER continues accessing objects and updating H and W_k. At any subsequent occasion when H becomes full again, its contents are sorted and merged with $\mathcal{Z}$; the resulting merged list $\mathcal{Z}$ is again written back to disk. We call Lines 8–9 a *refresh* operation because it derives *tight* lower score bounds for objects which have been seen (from any source). Observe that, between adjacent refresh operations, the lower bound score γ_x^{lb} derived at Line 4 is *loose*, since it does not consider any atomic scores which have been seen before and stored in $\mathcal{Z}$. After the growing phase ends, we store the candidates of H into $\mathcal{Z}$ and clean up H (Lines 12–14).

In the shrinking phase, we distinguish between two cases; (i) $\mathcal{Z}$ is empty (no candidates on disk) and (ii) $\mathcal{Z}$ is not empty. If $\mathcal{Z}$ is not empty and its contents do not fit in memory, the algorithm operates similarly to the growing phase; atomic scores are read from the inputs and a main-memory set of candidates is maintained in H, while W_k is updated. If H becomes full, we perform a refresh operation; H is merged with the set $\mathcal{Z}$ of candidates on disk and W_k is updated using the actual lower bounds. During merging, entries $y \in \mathcal{Z}$ with $\gamma_y^{ub} \leq t$ are deleted from $\mathcal{Z}$. Also, after each merging we apply the optimization of Section 5.3.2 to potentially dry up inputs (not shown in the pseudocode). Note that if $\mathcal{Z}$ is not empty, we need to store every accessed object in H, as we do not immediately check whether it has been seen before (we cannot use Lemma 3). As soon as the total number of candidates in $\mathcal{Z}$ fits in memory, they are loaded to the memory-resident H and case (i) applies. Since the set of candidates can only shrink, we construct the lattice and the algorithm operates like the normal shrinking phase of LARA, until termination.

We found by experimentation that the above disk-based algorithm incurs higher I/O cost in the growing phase than in the shrinking phase. The reason is that $\mathcal{Z}$ can grow very large and each refresh operation scans the whole $\mathcal{Z}$. To alleviate this problem, we propose a *lazy* approach which operates differently in the growing

```
Algorithm LARA-EAGER(memory bound $B$, ranked inputs $S_1, S_2, \ldots, S_m$)
1.  $\mathcal{Z}$:=new (empty) sorted list of objects on disk;
2.  repeat /* growing phase */
3.      access next object $x$ from next input $S_i$;
4.      update $x$ in $H$ and compute $\gamma_x^{lb}$; /* partial aggregate score */
5.      update $W_k$ if $\gamma_x^{lb} > t$;
6.      update $T$;
7.      if $|H| = B$ then /* memory full */
8.          union-merge the entries in $H$ and $\mathcal{Z}$ by object id, and write the merged list to $\mathcal{Z}$;
9.          during merging, update $\gamma_y^{lb}$ of objects $y \in \mathcal{Z}$ and update $W_k$;
10.         remove all entries from $H$;
11. until $t \geq T$;
12. if $\mathcal{Z} \neq \varnothing$ and $H \neq \varnothing$ then /* clean up candidates in memory */
13.     perform Lines 8–10;
14.     remove all entries with $\gamma_y^{ub} \leq t$ from $\mathcal{Z}$;
15. repeat /* shrinking phase */
16.     if $0 < |\mathcal{Z}| \leq B$ then
17.         load entries of $\mathcal{Z}$ into $H$, clear $\mathcal{Z}$, and construct lattice;
18.     access next object $x$ from next input $S_i$;
19.     if $\mathcal{Z} \neq \varnothing$ then /* entries on disk */
20.         update $x$ in $H$ and compute $\gamma_x^{lb}$;
21.         update $W_k$ if $\gamma_x^{lb} > t$;
22.         if $|H| = B$ then /* memory full */
23.             for each object $y \in \mathcal{Z}$ such that $y \in H$;
24.                 update $\gamma_y^{lb}$ in $\mathcal{Z}$ and $W_k$;
25.             remove all entries from $H$;
26.             remove all entries with $\gamma_y^{ub} \leq t$ from $\mathcal{Z}$;
27.     else /* no entries on disk */
28.         perform the shrinking phase of LARA;
29.         $u := \max\{\gamma_{x^v}^{ub} : v \in \mathcal{G}\}$; /* use lattice leaders */
30. until $\mathcal{Z} = \varnothing$ and $t \geq u$;
31. report $W_k$ as the top-$k$ result;
```

Fig. 7. Disk-based LARA-EAGER algorithm

phase. When H becomes full, we simply append new coming objects to $\mathcal{Z}$ and avoid scanning the whole $\mathcal{Z}$. After the growing phase terminates, we perform external sorting on $\mathcal{Z}$ by object id and then execute the refresh operation. LARA-LAZY can have lower I/O cost than LARA-EAGER. On the other hand, LARA-LAZY may have much higher access cost than LARA-EAGER, since the refresh operation is delayed until the end of the growing phase. During the growing phase of LARA-LAZY, W_k contains the highest lower bounds only from objects in H (i.e., the first H objects seen) and t is a loose bound. As a result, the termination of the growing phase delays.

In fact, there is a trade-off between accesses and I/O cost for disk-based implementations of LARA. LARA-EAGER performs refresh operations frequently, leading to low access cost and high I/O cost. On the other hand, LARA-LAZY delays refresh operations so it has low I/O cost but incurs many accesses. In order to achieve a good trade-off, we extend LARA-LAZY into a LARA-ADAPT algorithm, which performs refresh operations *adaptively* in the growing phase, aiming at balancing the access cost and disk I/O cost. Similar to LARA-LAZY, LARA-ADAPT simply appends new objects to $\mathcal{Z}$ after H becomes full. However, LARA-ADAPT differs from LARA-LAZY in the following points. Initially, a variable φ is set to the memory size B. Whenever $|\mathcal{Z}|$ reaches φ, LARA-ADAPT performs a refresh operation and doubles φ.[5] Thus, LARA-ADAPT is expected to (i) incur fewer I/O accesses than LARA-EAGER and (ii) perform fewer accesses than LARA-LAZY.

6. VARIANTS OF TOP-K SEARCH

So far we have discussed how LARA processes top-k queries when γ = `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 [Mamoulis et al. 2006], we have shown that LARA can easily be adapted for different variants and requirements of a top-k search. In this section, we propose adaptations of LARA for top-k join queries, top-k cube queries, and browsing operations between top-k cuboids.

6.1 Top-k *join* queries

The top-k query we have seen so far is a special case of top-k *join* queries [Natsev et al. 2001; Ilyas et al. 2003; Ilyas et al. 2004], where the results of joins are to be output in order of an aggregate score on their various component (see the discussion at Section 2.4). Here, we show 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, tuples that partially satisfy the join. Consider, for example, the top-k query expression of Section 2.4. 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 v, the highest upper bound $\gamma^{ub}_{x^v}$ for any partial tuple $x^v \in v$ is maintained as usual. This is computed by considering the last scores seen at any sources not in v. 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 scores. 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. The new tuple is accommodated in the corresponding lattice node and join results are immediately added to the corresponding nodes. Partial results in each lattice node are indexed in order to facilitate efficient probing (i.e., using hash tables).

The pseudocode of Figure 8 describes the functionality of LARA-J. The algorithm operates in a multiway ripple-join fashion, which has an integrated ranking component. Theorem 1 shows that it produces the correct results and only once (i.e., no duplicate join results are generated). The version of LARA-J shown in Figure 8 can be easily converted to an incremental algorithm, by replacing lines 9–10 by a while-loop that outputs the top tuple in $S_1 \circ S_2 \circ \cdots \circ S_m$ and removes it from the node, while its score is at least u.

[5]The rationale behind doubling φ is to result in total disk I/O that is approximately the sum of a geometric series (bounded by a constant factor of the number of accesses in the growing phase).

Theorem 1. *LARA-J produces the correct top-k join results and there are no duplicates in the final response set.*

Proof. We will first show that all complete multiway join results (i.e., if we exclude the top-k component of the query) are produced correctly and forwarded to the top lattice node only once. Let τ be a tuple in the multiway join result. Let x be the component of τ that arrived after all other components of τ in the lattice and assume that it arrived from source S_i. Clearly, τ will be generated at the lattice node only once and when x was read from S_i, since x would not have otherwise been known. In addition, the combination $\tau \backslash x$ of all other components in τ except x should be at lattice node $v = S_1 \circ \cdots \circ S_{i-1} \circ S_{i+1} \circ \cdots \circ S_m$ when x arrives. We can easily prove this assertion by induction, by considering the last arrived component x' of $\tau \backslash x$. The basis of the induction is that the temporally first component of τ (e.g., x_j that arrived from S_j) is trivially added once to the lattice node populated by the corresponding source (e.g., the node having S_j alone). In addition, the fact that tuples generated at any lattice node correspond to correct partial join results can also be easily proved by induction, since they are generated after probing the currently seen tuple to the contents of lattice nodes one level below. Finally, since upper bounds for partial results at all nodes are used, the ranking component of LARA-J is also correct. □

```
Algorithm LARA-J(ranked inputs $S_1, S_2, \ldots, S_m$)
1.  initialize lattice $\mathcal{G}$;
2.  access next tuple $x$ from next input $S_i$;
3.  add $x$ to node corresponding to source $S_i$;
4.  for each lattice node $v$ not containing source $S_i$
5.      join $x$ with the contents of $v$;
6.      add join results to node $v \circ S_i$;
7.      update upper bound of $v \circ S_i$, if applicable;
8.  $u := \max\{\gamma^{ub}_{x^v} : v \in \mathcal{G}\}$;
9.  $t :=$ $k$-th largest score at top node $S_1 \circ S_2 \circ \cdots \circ S_m$;
10. if $t < u$ then goto Line 2;
11. report tuples with the $k$-th largest scores at top node $S_1 \circ S_2 \circ \cdots \circ S_m$;
```

Fig. 8. The LARA-J algorithm

As already shown in [Ilyas et al. 2002], non-binary algorithms (like our LARA-J) could be more efficient than combinations of binary operators for problems with $m > 2$. LARA-J is expected to have several advantages over trees of binary operators like HRJN. First, LARA-J outputs the produced join results (produced at the top node) by comparing their scores with the upper bounds of the produced partial results at the lattice nodes. These upper bounds are expected to be lower than the thresholds used by HRJN (based on the best-case scores of potential future join results), so the output rate of LARA-J is expected to be higher than that of a plan of HRJN operators. In addition, the efficiency of the plan of HRJN operators is based on demand-driven evaluation, as the technique schedules $GetNext()$ calls at the various nodes of the operation tree. On the other hand, LARA-J adapts gracefully to production-based evaluation of top-k joins, as it produces all (complete and partial) join results as soon as a new tuple is available at any input. Finally, LARA-J does not rely on the generation and maintenance of an efficient plan of binary operators, but it produces the next join result with the highest score as soon as it can be guaranteed without any unnecessary accesses. The superiority of LARA-J compared to binary trees of HRJN is verified by our experimental evaluation.

6.1.1 *Reducing the number of lattice nodes in top-k joins.* Note that LARA-J joins each newly accessed tuple x with 2^{m-1} lattice nodes. However, not all these nodes may correspond to connected subgraphs of the complete multiway join graph. For instance, in the example query of Section 2.4, there is no join condition between R and T, thus lattice node R ∘ T essentially stores the Cartesian product of R and T. This implies that whenever a tuple is read from source R it will unconditionally be combined with all tuples at lattice node T and written to R ∘ T. Thus, the cost of writing to R ∘ T is quadratic to the number of tuples seen from R and T. In addition, space is wasted for this redundant Cartesian product. Finally, each tuple x read from S is joined with the very large Cartesian product stored in R ∘ T, while the join results it contributes to R∘S∘T are exactly determined by the tuples from R that join with x and the tuples from T that join with x (and these will be computed anyway). Thus, maintaining lattice nodes that correspond to disconnected subgraphs of the join graph leads to a waste of computational and storage resources.

We propose LARA-J*, an improved version of LARA-J which materializes only nodes of the lattice that correspond to connected query subgraphs. For this purpose, we model a multiway join query with a ranking component by a graph Γ, where nodes correspond to the joined relations and edges correspond to join predicates between attributes from the connected nodes.

The first difference between LARA-J and LARA-J*, is that the latter does not consider lattice nodes corresponding to sets of nodes that do not form a connected subgraph of Γ. The elimination of these nodes implies the second difference between the algorithms; special treatment is required for new tuples (e.g., from S), which were originally probed against a Cartesian product lattice node (e.g., R ∘ T) that now no longer exists. Not only do we have to generate partial results for non-eliminated nodes (e.g., R ∘ S and S ∘ T) having a join condition with the current source, but we may also need to generate tuples for other nodes of the lattice (e.g., R ∘ S ∘ T) that correspond to connected join (sub)graphs including the currently accessed source (e.g., S) and otherwise disconnected components (e.g., R and T). For our example join graph, when a tuple is accessed from R (T) it is joined with nodes S and S ∘ T (R ∘ S), as in LARA-J. On the other hand, a new tuple s that arrives from S is joined with nodes R and T only, but they produce results for R ∘ S, S ∘ T, and R ∘ S ∘ T. The tuples for the top node of the lattice are easily produced by taking the Cartesian product of the set of tuples from R and T that join with s; formally, the set $s \circ ((\mathtt{R} \ltimes \mathtt{s}) \times (\mathtt{T} \ltimes \mathtt{s}))$, where $\ltimes$ denotes a semi-join. For instance, if s joins with $\{r_1, r_2\}$ from R and $\{t_1\}$ from T, $\{r_1, s, t_1\}$ and $\{r_2, s, t_1\}$ are added to node R∘S∘T without the need of extra computations.

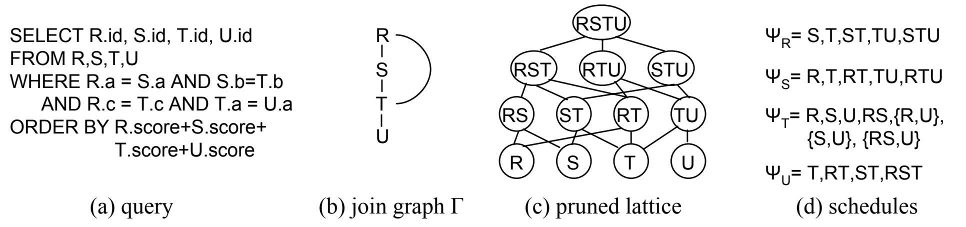
Thus, the handling of newcoming tuples varies between different sources, depending on the corresponding nodes of the join graph. In order to avoid re-computing from scratch the set of lattice nodes that are affected and the corresponding operations for every tuple, LARA-J* includes a *query preprocessing* phase, where for every source S_i a *schedule* Ψ_i of operations to lattice nodes is generated. This schedule is static throughout the join evaluation and it is computed beforehand. Every time a tuple is read from S_i, a sequence of operations are performed to lattice nodes as indicated by Ψ_i. Figure 9 shows a pseudocode for this preprocessing step. First, lattice nodes corresponding to disconnected subgraphs of Γ are eliminated. Then, nodes of the lattice that do not contain S_i, but they are connected to S_i in the join graph, are added to schedule Ψ_i. A single lattice v node in Ψ_i indicates that a new tuple that arrives from S_i will be probed against the contents of v (partial tuples) and the resulting tuples will be promoted to node $v \circ S_i$. The final step of the preprocessing step is to identify the lattice nodes where Cartesian products of the computed join results will be inserted. For each combination of lattice nodes $\{v_1, v_2, \ldots, v_l\}$ in Ψ_i which are disjoint (in pairs) in Γ, LARA-J* should generate tuples in nodes $v_1 \circ v_2 \circ \cdots \circ v_l \circ S_i\}$, by taking the Cartesian products of the new tuples at each $v_j \circ S_i$.

```
Algorithm LARA-J*-prep(ranked inputs $S_1, S_2, \ldots, S_m$, join graph $\Gamma$)
1.  initialize lattice $\mathcal{G}$;
2.  eliminate nodes from $\mathcal{G}$ corresponding to disconnected subgraphs of $\Gamma$;
3.  for each input $S_i$
4.      for each node $v$ of $\mathcal{G}$, such that $S_i \notin v$ and $v$ is connected to $S_i$ in $\Gamma$
5.          add $\{v\}$ to $\Psi_i$;
6.  for each combination $\{v_1, v_2, \ldots, v_l\}$ of lattice nodes in input $\Psi_i$
7.      if $\forall x, y \in [1, l], sources(v_x) \cap sources(v_y) = \varnothing$ and
                $\forall x, y \in [1, l], sources(v_x) \cup sources(v_y)$ form a disconnected subgraph in $\Gamma$ then
8.          add $\{v_1, v_2, \ldots, v_l\}$ to $\Psi_i$;
```

Fig. 9. The preprocessing phase of LARA-J*

Figure 10 shows an example top-k join query, the corresponding join graph, the nodes of the lattice that remain and the schedule Ψ_i for each source S_i. Note that source T divides connected subgraph {R,T,U} into disconnected components ({R} and {U}); the combination of these components is included it its schedule. T, if removed, also splits {S,T,U} and {R,S,T,U} to disconnected components. Thus ({S},{U}) and ({R $\circ$ S},{U}) are added to $\Psi_{\mathtt{T}}$.

The pseudocode of LARA-J* is described in Figure 11. First, the preprocessing phase is called to initialize and prune the lattice $\mathcal{G}$ and generate a schedule for each source. Then, tuples are accessed from the various inputs and the necessary join



```
SELECT R.id, S.id, T.id, U.id
FROM R,S,T,U
WHERE R.a = S.a AND S.b=T.b
    AND R.c = T.c AND T.a = U.a
ORDER BY R.score+S.score+
        T.score+U.score
```

Fig. 10. Example of LARA-J* preprocessing

operations are performed. For each tuple x accessed from S_i, first x is added to node S_i and then it is probed against the contents of the single nodes in Φ_i (i.e., not node combinations) to generate and promote partial join results. Finally, for each combination of lattice nodes that appears in Ψ_i the Cartesian product of the new join results (due to x) is generated and promoted to the corresponding lattice node. For the example query of Figure 10, if a tuple x is accessed from source T, first x is probed against the contents of nodes R, S, U, and R $\circ$ S. The new join results are promoted to the corresponding nodes that include T, in addition to the other sources. Finally, the Cartesian products of the new results produced at R $\circ$ T and T $\circ$ U is computed and inserted to R $\circ$ T $\circ$ U; the same process is applied for the remaining composite components of Φ_i, i.e., ({S},{U}), and ({R $\circ$ S},{U}). Like in LARA-J, the contents of the top node are organized in a priority queue and output incrementally as they are found greater than the upper bounds of the remaining nodes. Theorem 2 proves the correctness and completeness of LARA-J*.

Theorem 2. *LARA-J* produces the correct top-k join results and there are no duplicates in the final response set.*

Proof. The proof is based on that of Theorem 1. The difference between the algorithms is due to the elimination of lattice nodes that correspond to Cartesian products. In other words, we only need to prove that the contents of such nodes (which are not materialized) are considered in the generation of tuples at levels above them. This is ensured by the schedule list of each source. In specific, let y be a tuple to be generated at node v by joining the currently read tuple x from S_i with the contents of node $v \backslash S_i$ (i.e., containing all sources of v but S_i). Assume that $v \backslash S_i$ is not materialized because it corresponds to a Cartesian product. In that case, the set of connected subgraphs in the join subgraph containing the nodes of $v \backslash S_i$ will be in the schedule Ψ_i corresponding to source S_i. The contents of this non-materialized node that should be joined with x are generated on-demand by LARA-J*, as discussed, so no join results will be missed. □

```
Algorithm LARA-J*(ranked inputs $S_1, S_2, \ldots, S_m$, join graph $\Gamma$)
1.  LARA-J*-prep($S_1, S_2, \ldots, S_m, \Gamma$);
2.  access next tuple $x$ from next input $S_i$;
3.  add $x$ to node corresponding to source $S_i$;
4.  for each single lattice node $v$ in $\Psi_i$
5.      join $x$ with the contents of $v$;
6.      add join results to node $v \circ S_i$;
7.      update upper bound of $v \circ S_i$, if applicable;
8.  for each combination of lattice nodes $\{v_1, v_2, \ldots, v_l\}$ in $\Psi_i$
9.      generate Cartesian product of newly generated results from each $v_j \circ S_i$
              and promote them to node $\{v_1 \circ v_2 \circ \cdots \circ v_l \circ S_i\}$;
10. $u := \max\{\gamma^{ub}_{xv} : v \in \mathcal{G}\}$;
11. $t :=$ $k$-th largest score at top node $S_1 \circ S_2 \circ \cdots \circ S_m$;
12. if $t < u$ then goto Line 2;
13. report tuples with the $k$-th largest scores at top node $S_1 \circ S_2 \circ \cdots \circ S_m$;
```

Fig. 11. The LARA-J* algorithm

6.2 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.

6.2.1 *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 5.3.2.

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. The effect of this optimization is experimentally studied in [Mamoulis et al. 2006].

6.2.2 *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 [Natsev et al. 2001; Ilyas et al. 2002; 2003; Ilyas et al. 2004]. 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. In Section 7, we compare an implementation of LARA for complex aggregate functions to a simple implementation of NRA.

6.2.2.1 *Multiple leaders in complex aggregate functions.* The computational efficiency of LARA in the shrinking phase is achieved by maintaining a leader object x^v at each lattice node v, which helps to prune all objects currently seen at the combination of inputs in v. Recall that x^v is the object with the highest *partial* aggregate score seen only in v, but currently not in W_k.

The above technique requires the monotone aggregate function γ to be *complete subset-decomposable*, i.e., for any subset v of sources, there exist monotone aggregate functions f_v and g_v, such that $\gamma(x) = g_v(f_v(x), x)$ where (i) inputs of f_v come only from atomic scores of x in v, and (ii) inputs of g_v come from $f_v(x)$ and atomic scores of x not in v.[6] If γ is complete subset-decomposable, then the partial aggregate f_v defines a *total* order for objects within the same node, regardless of the latest values seen from other sources. Simple functions such as (weighted) `sum`, `min`, and `max` belong to this class of functions. This means that we can compute a *partial* score for each seen object and use these partial scores to define unique leaders for lattice nodes. In addition, the leader of a node v cannot change, unless it is promoted to another node or a new object is promoted to v.

Nevertheless, some complex aggregate functions are not complete subset-decomposable, which means that we can find lattice nodes v, for which we cannot define monotone functions f_v and determine unique partial scores for the objects seen only there. For such nodes, we cannot define unique leaders; i.e., the upper bounds of objects there do not define a fixed total order. As an example, consider function $\gamma = \min\{S_1, S_2\} + \min\{S_3, S_4\}$ and node $v = S_2S_3$. Assume that v contains two objects $a = (?, 0.5, 0.7, ?)$ and $b = (?, 0.6, 0.4, ?)$. For these two objects, it is necessary to keep both atomic scores in order to derive their upper bound. In addition, a and b are not dominated by each other.[7] As a result, the object with the highest upper bound (i.e., the leader of v), depends on the last values seen at S_1 and S_4. For example, if $l_1 = l_4 = 0.75$, then $\gamma_a^{ub} > \gamma_b^{ub}$. However, if $l_1 = 0.65$ and $l_4 = 0.35$, then $\gamma_b^{ub} > \gamma_a^{ub}$. Thus, we cannot define a unique leader for v that is guaranteed to remain unchanged if the contents of v remain constant.

A possible solution to this problem is to maintain the *skyline* of leaders [Börzsönyi et al. 2001] at each problematic node of a complex aggregate. Nevertheless the skyline can be very large even for 2 dimensional datasets, leading to an uncontrolled number of leaders per node and possible explosion of LARA's computational cost. Besides, the maintenance of skylines is expensive. Instead, we suggest a variant of the LARA-MAT algorithm (proposed in Section 5.3.1), which explicitly maintains the set of objects associated with the lattice nodes. Let C_v be the set of (candidate)

[6]The subset-decomposable function is a generalization of the distributive monotone function [Gray et al. 1997]. For distributive monotone functions γ, we have $\gamma = g_v = f_v$.

[7]A multidimensional object x dominates object y, if the atomic scores of x are no smaller than the corresponding atomic scores of y. The concept of dominance is used in skyline queries [Börzsönyi et al. 2001].

objects (not in W_k) associated with node v. For each node v, which has non-unique leader, we pick a random object in C_v and use it as *approximate* leader. The selected leader is kept fixed, unless it is promoted or it gets pruned. Whenever the upper bound score of an approximate leader is below t, there exists some objects in C_v (at least one) that can be pruned. Thus, we scan C_v and remove objects having upper bound score below t. From the remaining objects, a random object is picked as the new approximate leader for the node of v. Eventually, the node v is marked empty after all its associated objects have been pruned.

This version of LARA-MAT is expected not to have high computational overhead compared to the application of the algorithm on nodes with unique leaders. Each time we pick an approximate leader at a node its expected upper bound is the median of the upper bounds of nodes in C_v. Thus, each time the leader is used to prune C_v, its size shrinks by half. Assuming that C_v remains constant during LARA-MAT, the total number of objects examined until it is completely pruned is: $|C_v|(1 + 1/2 + 1/4 + \cdots)=2|C_v|$. This cost is acceptable when compared to the lower bound cost $|C_v|$ of examining and pruning all objects (i.e., if the approximate leader happens to be the one with the largest upper bound). Finally, note that a more principled approach, which would pick/maintain the best leader (based on the number of objects it dominates in C_v), is expected to have higher computational cost.

6.3 Top-k Cubes

In this section, we investigate an interesting subset of OLAP queries that involve ranking and design effective extensions of LARA for them. Consider a set of m ranked inputs and assume that we are interested to retrieve the top-k objects in *every* combination of these sources. The result of this operation is called the *top-k cube*. For instance, the top-1 cube for the data of Figure 1, (assuming $\gamma = \texttt{sum}$) is $\{\langle S_1, c\rangle\ \langle S_2, a\rangle\ \langle S_3, c\rangle\ \langle (S_1, S_2), b\rangle\ \langle (S_1, S_3), c\rangle\ \langle (S_2, S_3), a\rangle\ \langle (S_1, S_2, S_3), b\rangle\}$. Data analysts could use the top-k cube to compare objects that appear in top-k results of different combinations of sources. For instance, if a particular object appears in multiple top-k sets it can be considered globally important for different users that search for top-k results by combining different ranking attributes. The top-k cube can also be used to identify correlations between attribute sets, if they share many common top-k results.

A brute-force method for the computation of a top-k cube from m ranked inputs is to apply LARA (or NRA), iteratively; once for every combination of sources. The computational cost and the required accesses of such an approach is high. In addition, we might not be able to apply this method if the sources are allowed to be accessed only once (e.g., they come as pipelined results of other database operations).

A more suitable technique is to run an NRA algorithm only once and compute the top-k set for each combination v of sources simultaneously. The simple NRA algorithm can be adapted to an NRA-CUBE method, which maintains for each combination of sources (corresponding to a lattice node v), a set W_k^v; the k objects with the highest ${}_v\gamma^{lb}$, where ${}_v\gamma$ denotes the result of the aggregate function γ when considering only the set of sources v. In order for NRA-CUBE to terminate, for every W_k^v there should be no object $x \notin W_k^v$, such that ${}_v\gamma_x^{ub} > t^v$, where t^v is the

S_1	S_2	S_3
a 0.9	b 0.9	b 0.9
f 0.2	a 0.7	e 0.3
...	...	...

Fig. 12. Top-k cubing example

k-th score in W_k^v. This check is much more expensive compared to the termination check of the simple NRA algorithm.

We now investigate the essential changes to be performed on LARA, in order to come up with LARA-CUBE, an efficient top-k cube algorithm.

6.3.1 *The growing phase.* Let $T_v = \gamma\{l_i : i \in v\}$, i.e., the result of the aggregate function when applied to the last values seen in the sources that appear in a combination v. Let W_k^v be the k objects with the highest ${}_v\gamma^{lb}$ at any instance of the algorithm and t^v be the k-th score in W_k^v. Similarly to the basic version of LARA, we cannot prune any object until $t^{ALL} \geq T^{ALL}$, where ALL is the combination $S_1 S_2 \ldots S_m$ of all sources. Furthermore, even when $t^{ALL} \geq T^{ALL}$, if there is a $v \neq ALL$, such that $t^v < T^v$, any object not seen at all yet could end up in the top-k set of v. Figure 12 illustrates an example. Assume $k = 2$ and $\gamma =$`sum`. After two rounds of accesses, $W_k^{ALL} = \{(b, 1.8), (a, 1.6)\}$ and $t^{ALL} > T^{ALL} (= 1.2)$. However, $W_k^{S_2 S_3} = \{(b, 1.8), (a, 0.7)\}$ and $t^{S_2 S_3} < T^{S_2 S_3} (= 1.0)$, which means that objects not seen yet could enter the top-k set of combination $S_2 S_3$. This example shows that we should not terminate the growing phase until $t^v \geq T^v$ for all v and no monotonicity property holds among combinations (i.e., the combination $S_2 S_3$ does not satisfy $t^v \geq T^v$ although its subsets and supersets do).

In our example, $t^v \geq T^v$ for all v, except $S_2 S_3$. In order to avoid accessing more objects than necessary, we can *stall* the accesses to source S_1, until the condition $t^{S_2 S_3} \geq T^{S_2 S_3}$. Stalling accesses to one or more sources during the growing phase of top-k cube computation is similar to the "drying up" of sources (see Section 5.3.2); stalling of S_i is applied as soon as the condition $t^v \geq T^v$ holds for all v that include S_i. When LARA-CUBE enters the shrinking phase, we resume accesses to the stalled sources.

6.3.2 *The shrinking phase.* The shrinking phase of the basic LARA algorithm creates the virtual lattice by holding for each combination the object seen exactly in these sources with the highest upper bound that does not currently appear in W_k. A direct extension of this idea for the shrinking phase of top-k cube queries is to create a separate lattice for each of the $2^m - 1$ aggregate functions and for each function create/update a W_k^v set and a $2^{arity(v)}$ set of leaders. After each access — e.g., object x from source S_i — the W_k^v set of each combination v that includes S_i and its respective leaders are potentially updated. Once the W_k^v for a combination v is finalized, the corresponding lattice and leaders are deleted. If at some stage a source S_i cannot contribute to any W_k^v, then the source is dried up (as in Section 5.3.2).

For example, consider the lattice of Figure 4. For combination $v = S_1 S_2$, we keep (i) $W_k^{S_1 S_2}$; the set of k objects with the highest ${}_{S_1 S_2}\gamma^{lb}$; (ii) leaders at nodes S_1 and S_2 with respect to combination $S_1 S_2$ and function ${}_{S_1 S_2}\gamma$. Thus, an object x seen at

S_1 and S_3 could be a leader at node S_1 for the top-k query of combination S_1S_2; in the basic LARA algorithm x could only be a leader at S_1S_3. To determine whether $W_k^{S_1S_2}$ is the exact top-k set for function ${}_{S_1S_2}\gamma$, $t^{S_1S_2}$ (that is the k-th object in $W_k^{S_1S_2}$) must be not smaller than the upper bounds of all S_1S_2-related leaders for $v' \in \{S_1, S_2\}$.

The above technique requires much more bookkeeping compared to the basic LARA algorithm, since we need to maintain and update $\sum_{l=1}^{m} \binom{m}{l} 2^l$ instead of $O(2^m)$ leaders. To reduce this computational burden, in the shrinking phase, we avoid processing the query for all source combinations *concurrently*, but compute the top-k results of combinations *iteratively*, in breadth first order, starting from the topmost combination. Consider, for example, a top-k cube query on three inputs $S_1S_2S_3$. In the shrinking phase of LARA-CUBE, we first consider the topmost combination $v = S_1S_2S_3$, by constructing the lattice for v, and running the basic LARA (without pruning objects) until the top-k results of v are computed. Objects are not pruned because they could appear in top-k results of other combinations (e.g., $\{S_1, S_3\}$) that will be later processed. To minimize accesses, a source S_i is dried up (see Section 5.3.2) when all qualified candidates for v (those with upper bound score above t^v) have been seen from S_i. After the top-k results of $S_1S_2S_3$ are computed, we *continue* to the next combination (e.g., $v' = S_1S_2$) from the state where the computation has stopped. Thus, we scan the hash table to update $W_k^{v'}$, and construct the lattice for v', computing leaders (based on ${}_{v'}\gamma$) for its nodes S_1 and S_2. Accesses are continued from inputs S_1 and S_2 (if necessary) until the top-k result (based on ${}_{v'}\gamma$) is finalized. LARA-CUBE, continues in this manner and computes iteratively the results for the remaining combinations (i.e., $\{S_2, S_3\}$, $\{S_1, S_3\}$, $\{S_1\}$, $\{S_2\}$, and $\{S_3\}$). This implementation of LARA-CUBE saves numerous computations as leaders and top-k sets for different combinations are not maintained concurrently.

6.4 Browsing Through Top-k Results at Different Dimension-Sets

Similar to browsing operations in OLAP systems, a user may wish to navigate through the top-k results of various combinations of ranked lists that can be aggregated. For instance, consider a user who has at hand the top-10 restaurants in terms of price, quality, and distance to her location. Being not satisfied by these results she may wish to *roll-up* to the list of top-10 restaurants in terms of price and distance only, or she may want to *drill-down* to the top-10 set in terms of price, quality, distance, and size.

Formally, given a top-k query on a set $\mathcal{S}$ of m ranked inputs, rolling-up (drilling-down) refers to the operation of retrieving the top-k results by applying the same γ function to any subset (superset) of $\mathcal{S}$. As we show in the following, LARA allows the derivation of the new top-k set fast, subject to certain (minor) changes applied to the original algorithm. Our roll-up and drill-down techniques save significant access cost and computational time, over running the query from scratch.

6.4.1 *Rolling Up.* Assume that the set of top-k objects, aggregated using a γ function over m sources $\mathcal{S}$, has been finalized. Assume that the user requests the top-k objects with respect to only a subset $v \subset \mathcal{S}$ of sources. If, during the shrinking phase of the previous query, LARA has not pruned any object,[8] we can avoid processing the new query from scratch. The main idea is to move the original top-k problem from the top node of the original query's lattice to the node v of the roll-up query. For this, the following steps are applied:

—Compute W_k^v by accessing the objects that are assigned to all lattice nodes (of the original query) which have at least one source common with v. For all these objects ${}_v\gamma^{lb}$ should be computed.

—Compute the leader objects at all nodes of the original lattice that are descendants of v.

—Terminate if $t^v \geq \max\{{}_v\gamma^{ub}_{x^{v'}} : v' \subset v\}$, where $x^{v'}$ is the new leader of node v' (with respect to ${}_v\gamma^{ub}$ and W_k^v). Resume a projected version of LARA on the v sources and the constrained lattice to subsets of v, otherwise.

For example, consider the lattice at Figure 4 for the top-1 `sum` query on the data of Figure 1. Figure 4b shows the contents of the lattice when LARA terminates returning b with score 2.2. Assume that the user at this stage requests the object with the highest `sum` of atomic scores at sources S_1 and S_2 only. The incremental roll-up LARA module first computes $W_k^{S_1S_2}$ by accessing the objects seen at source combinations $\{S_1, S_2, S_1S_2, S_1S_3, S_2S_3, S_1S_2S_3\}$ and computing their ${}_{S_1S_2}\gamma^{lb}$. This results to $W_k^{S_1S_2} = b$, with ${}_{S_1S_2}\gamma_b^{lb} ={}_{S_1S_2}\gamma_b = 1.4$. Then leaders for nodes S_1 and S_2 are computed; these are c and a, respectively, with ${}_{S_1S_2}\gamma_c^{ub} = 1.3$ and ${}_{S_1S_2}\gamma_a^{ub} = 1.2$. Since both leaders have lower upper bound than b's score, the algorithm terminates reporting b. In this example, no additional accesses are required by the roll-up operation.

6.4.2 *Drilling Down.* For drill-down top-k operations, we also assume that LARA has not pruned any object during the shrinking phase of the previous query. Similar to rolling-up, our objective is to avoid starting LARA from scratch, but to move the original top-k problem from the top node of the original query's lattice to the new top node after introducing new sources and extending the current lattice by all combinations with the existing ones. More specifically, the following steps are applied:

—Extend the lattice to include the new sources and combinations.

—Compute W_k for the new top node by applying the γ^{lb} function to the top-k set of the previous query (no object outside this set could be in the top-k of the new query without any access to the new sources).

—Keep the leaders of the original lattice nodes; they do not have to be revised, since we start with the W_k of the previous query.

—Resume LARA by accessing objects from the old and the new sources.

Again, consider the lattice at Figure 4b after the termination of the top-1 `sum` query on the data of Figure 1. Consider a fourth source S_4 and assume that we want to determine the object with the highest sum of atomic scores at all four inputs. First, we extend the lattice to include all combinations of sources that include S_4.

[8]otherwise top-k results of dimensional subsets may be lost as discussed in Section 6.3

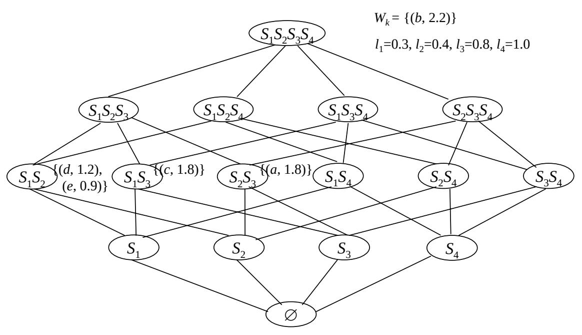


Fig. 13. Lattice derived by a drill-down top-k query

The existing $W_k = \{(b, 2.2)\}$ now refers to the combination of all four sources. Thus $t = 2.2$ remains valid. The revised lattice is shown in Figure 13. The existing leaders at nodes S_1S_2, S_2S_3, and S_2S_3 remain valid. The termination condition does not hold after the inclusion of source S_4, since the upper bound for c (who is leader in S_1S_3) is $\gamma_c^{lb} + 0.4 + 1.0 = 3.2 > t$ (we assume that the scores in S_4 range from 1 to 0). LARA continues by accessing objects from all four sources, until the top-k result is finalized. Note that this incremental drill-down operation has minimal cost, since no computations are required to update leaders or W_k before we resume the accesses. On the other hand, rolling up requires scanning the objects seen at any of the sources of the new query and updating W_k and the leaders.

7. 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, top-k join, 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 D 2.8GHz PC with 1GB of RAM.

7.1 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 [Börzsönyi et al. 2001]. 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 bad 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 the following real dataset from the UCI KDD Archive.[9] FC contains a set of 581,012 objects, corresponding to 30×30-meter forest land cells. Each object is described by various variables, e.g., distances to hydrology, roadways, fire points, etc. 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 values of each attribute were normalized in the range between 0 and 1. For some attributes, the scores were reversed (by subtracting them from 1) in order for 1 to indicate high preference and 0 low preference. In [Mamoulis et al. 2006], the effectiveness of LARA is also evaluated on another real dataset with similar results.

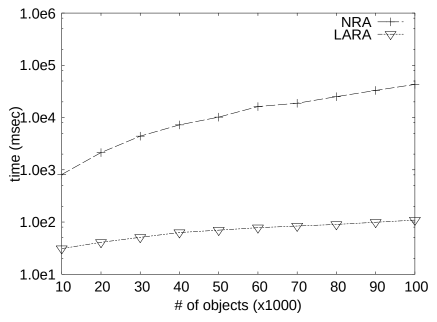
7.2 Experiments for classic top-k queries

In the first set of experiments, we compare LARA with the NRA algorithm of [Fagin et al. 2001] 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.

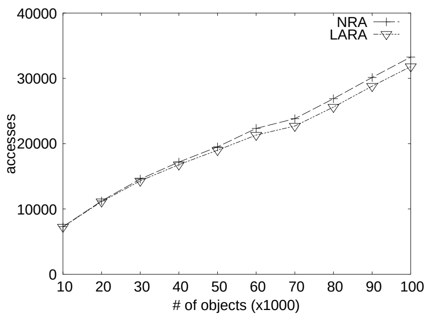
Figures 14a, 15a, and 16a compare the efficiency (in CPU time) of the two methods on uniform data (UI), for a range of parameter values. The default values for the parameters are n = 50K, $k = 20$, and $m = 3$. In each experiment, we fix two parameters to their default values and vary the value of the third one. In Figures 14a and 15a, LARA is about 2 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 $O(|C|)$ required by NRA. This explains the increase of performance gap with the increase of n. On the other hand, the difference is insensitive to k, as shown in Figure 15a.

LARA outperforms NRA in terms of the number of object accesses, as well, but the difference is marginal. Indicatively, Figure 14b shows the number of object accesses on uniform data by both methods as a function of n. LARA's optimization described in Section 5.3.2 saves 1%–5% of NRA's accesses, because of "dried up" streams toward the end of the algorithm. As shown in Figure 15b, the difference in accesses is similar when the parameter k changes. As we will see later, LARA may accesses significantly fewer objects than NRA in top-k queries on real data, where

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

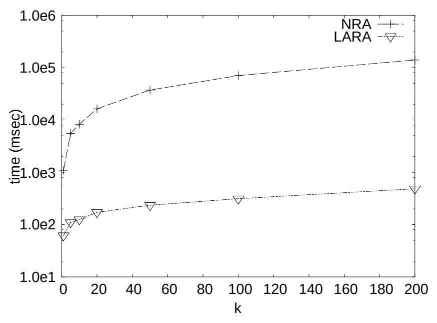


(a) time

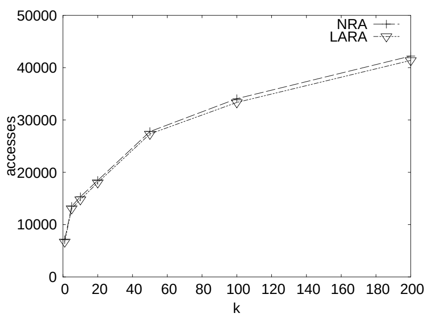


(b) accesses

Fig. 14. Effect of n on top-k queries ($\gamma = \texttt{sum}$, UI, $m = 3$, $k = 20$)



(a) time

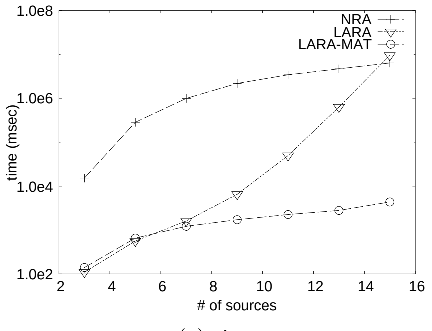


(b) accesses

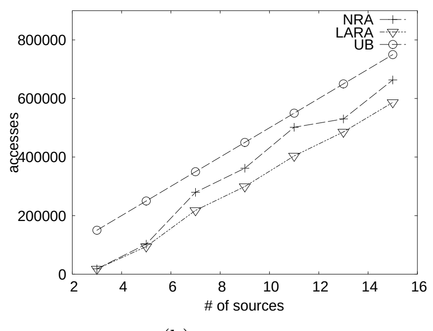
Fig. 15. Effect of k on top-k queries ($\gamma = \texttt{sum}$, UI, $n = 50$K, $m = 3$)

the distribution of scores in different inputs varies significantly.

Figure 16a plots the CPU time with respect to the number of sources m. As m grows, the size of the lattice increases exponentially and the CPU time difference between NRA and LARA shrinks. For large values of m, the number of objects at each lattice node becomes small and it becomes more likely for a leader to be promoted. In addition, the total number of leaders (2^m) greatly increases with m. For values of m above 15, the computational cost of LARA exceeds that of NRA. For high values of m, LARA-MAT, the alternative implementation of LARA discussed in Section 5.3.1, turns out to be more efficient and scalable. Its performance is also plotted in Figure 16a. LARA-MAT *materializes* the lattice by explicitly maintaining for each node the set of candidates seen at the corresponding set of inputs. In this implementation, leader updates do not require the scanning of the whole candidate set, but only the objects currently in the node where the leader should be updated. In addition, LARA-MAT keeps track of the subset of nodes for which the leader (and remaining objects) have not been pruned. This number is much smaller than 2^m, and as a result LARA-MAT scales very well for large values of m, maintaining its advantage over NRA. The overhead of LARA-MAT compared to LARA is that at each access (e.g., of object x), the contents of two

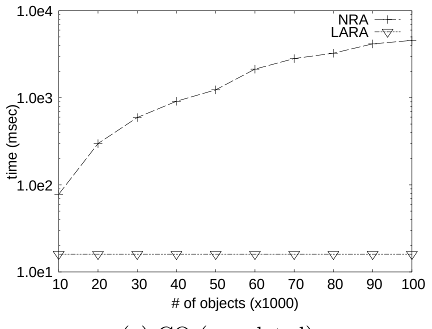


(a) time

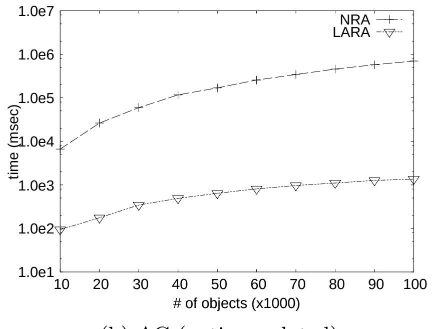


(b) accesses

Fig. 16. Effect of m on top-k queries (γ =**sum**, UI, n = 50K, k = 20)



(a) CO (correlated)



(b) AC (anti-correlated)

Fig. 17. Time of top-k queries on non-uniform data (γ =**sum**, $m = 3, k = 20$)

nodes must be updated (e.g., x is deleted from v_x^{prev} and inserted to v_x). Note that this overhead does not pay off at low values of m, where leader updates are very rare and there are few nodes in the lattice. Figure 16b shows the number of accesses as a function of m. In the graph, we also include UB, the access cost of reading all scores from all sources. Observe that LARA has fewer accesses than NRA for all values of m. However, as m increases, the performance gap between UB and NRA/LARA shrinks, due to the dimensionality curse [Beyer et al. 1999]. This figure also confirms that top-k retrieval on uniform data is not meaningful for large m.

The performance gap between NRA and LARA is similar for correlated and anti-correlated data (Figures 17a and 17b). 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 (i) the number of candidates $|C|$ increases with n and (ii) the cost of NRA is O($|C|$) (refer to the space and time complexity analysis of Section 5). 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. The value of n has a smoother effect on LARA, which retrieves 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 seven rankings of the objects according to their horizontal distance to hydrology (hh), vertical distance to hydrology (vh), horizontal distance to roadways (hr), horizontal distance to fire points (hf), morning hill-shade (sa), noon hill-shade (sn), and afternoon hill-shade (sp). 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 18 summarizes the results. NRA performs 23% to 65% more accesses than LARA. As illustrated in Section 5.3.2, this is attributed to the distribution of the scores which is irregular in some sources (e.g., sa); 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. Observe that LARA is 3 to 4 orders of magnitude 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 bookkeeping 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 19 shows the number of objects x with $\gamma_x^{ub} \geq t$ during the execution of LARA for the top-k queries of Figure 18. The x-axis is the ratio of accesses until termination. The plot shows that in all queries the candidates grow linearly with the number of accesses during the growing phase, and there is a sharp drop during the shrinking phase (especially for smaller values of m). The memory requirements of LARA (and NRA algorithms in general) are high in queries of higher dimensionality (more than 50% of the objects become candidates during the growing phase for $m = 5$ and $m = 6$). This result is consistent with our space analysis and with our argument that the $O(|C|)$ per-access cost of NRA can be very large, compared to LARA's $O(\log k)$ and $O(\log k + 2^m)$ costs in the growing and shrinking phases, respectively. An interesting observation from the diagram is that at least 70% of the accesses in the shrinking phase are spent to verify whether the very few remaining candidates can be part of the top-k result. This indicates that a mixed strategy (i.e., computing the scores of remaining candidates by random accesses if these drop to a very small percentage of n) could be more beneficial than a pure NRA algorithm. If random accesses are not possible, one could resort to an *approximate* technique, which returns all candidates if their number is a small multiple of k. A detailed study of approximate top-k NRA methods with quality guarantees is out of the scope of this paper, but remains an interesting subject for future work.

7.3 Experiments for top-k search with bounded memory

We performed an experiment to validate the effectiveness of LARA for cases where the set of candidates does not fit in memory (see Section 5.4). We set the disk page size to 1K bytes, such that each page can hold up to 100 objects with their partial scores. Figure 20 summarizes the access cost and disk I/O cost (in terms of disk page accesses) of the algorithms as a function of the memory bound B, expressed

attributes of FC	number of accesses		time in seconds	
	NRA	LARA	NRA	LARA
$\{hh, hr, sa\}$	1047027	639400	1473 (560)	1.8 (0.5)
$\{hh, vh, hr, sn\}$	1875047	1297259	5514 (3964)	4.1 (1.4)
$\{hh, hr, hf, vh, sp\}$	1573173	980098	9004 (6840)	4.5 (1.8)
$\{hh, hr, hf, vh, sa, sp\}$	2867548	2323667	37493 (27228)	13.4 (3.6)
$\{hh, hr, hf, vh, sa, sn, sp\}$	3345486	2531264	39276 (29067)	19.6 (3.7)

Fig. 18. Forest coverage (γ =`sum`, $k = 20$)

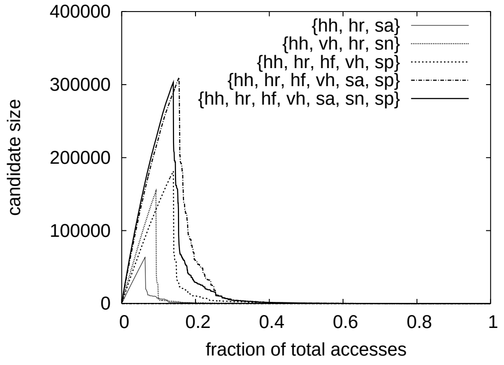


Fig. 19. Evolution of top-k candidates during LARA for the queries of Figure 18

as a percentage of the required memory to store all n objects and their scores. The result confirms our observation in Section 5.4. For small memory bounds, LARA-EAGER and LARA-LAZY have high I/O cost and high access cost respectively. When compared to LARA-EAGER and LARA-LAZY, the performance of LARA-ADAPT is less sensitive to the memory size and the algorithm achieves a good balance between access cost and disk I/O cost. Note that the access cost of LARA-ADAPT (and LARA-EAGER) is very close to the minimum possible (i.e., the cost of LARA for unbounded memory), even for very low values of B. Summing up, we recommend LARA-ADAPT as the most appropriate algorithm for top-k search with bounded memory.

7.4 Experiments for top-k joins

We compare the top-k *join* version of LARA (i.e., LARA-J, LARA-J*) with a tree of binary HRJN operators [Ilyas et al. 2003]. We joined three relations `R`, `S`, and `T`, of the same schema: (`id`, `score`, `j`). `j` is the attribute with respect to which *all* 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` $\bowtie$ `S`) $\bowtie$ `T` (other plans have similar performance). Figure 21 plots the number of tuples accessed by HRJN, LARA-J, and LARA-J* 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

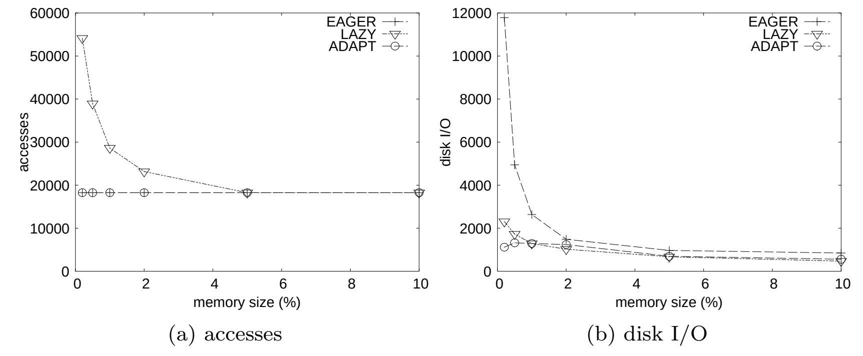


Fig. 20. Effect of memory size on top-k search (γ =sum, $n = 50$K, $m = 3$, $k = 20$)

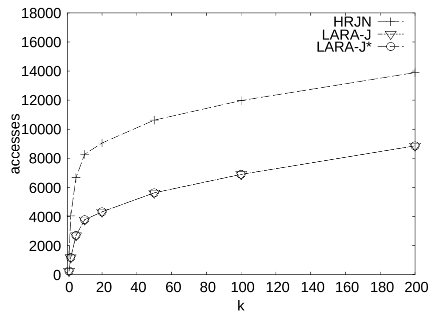


Fig. 21. Top-k join with complete join graphs, (UI, $n = 50\text{K}, m = 3$)

specific number of accesses). Observe that LARA-J (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 (LARA-J is as efficient as HRJN, at no expense of accessing more objects than necessary) but also indicates that multiway top-k operators can be more effective (in terms of accesses) than trees of binary join operators.

For the query in the last experiment, the optimized LARA-J* algorithm proposed in Section 6.1.1, has identical performance to that of LARA-J, since the join edges form a complete graph. In the following experiment, we compare LARA-J, LARA-J*, and HRJN for complex top-k join queries with graphs Γ that are not complete. We generated four relations `R`, `S`, `T`, `U`, each ordered in descending order of their `score` attribute. The score of each tuple was uniformly generated in the range $[0, 1]$. Each relation also has three additional attributes `a`,`b`,`c`, used in join conditions. The values of these attributes were randomly generated integers in [1,100]. Finally an `id` (primary key) was generated for each tuple. The cardinality of each relation is $n = 50000$. Figure 22 compares LARA-J, LARA-J*, and the best HRJN plan (i.e., the one with the fewer accesses). The comparison includes computational cost, number of accesses, and the total number of intermediate results when each method terminates. The last figure corresponds to the maximum memory required until the top-k join result is output. Note that we implemented and compared incremental versions of all algorithms, which means that no partial join results are pruned during join evaluation.

As the figure shows, LARA-J* has significant improvement over LARA-J, in terms of computational cost and required memory. The difference is because of two reasons. First, Cartesian products are not computed and materialized in LARA-J*, which apart from the large space savings, implies savings also in computational time. Second, joins with Cartesian products are avoided and replaced by the dynamic production of the results of such joins, for each incoming tuple x. That is, for each Cartesian product (e.g., $A \times B$) to be joined with x, there is an entry in the schedule of the input S_i, where x arrived from (e.g., $(\{A\}, \{B\})$). The join results which contain x and tuples from the Cartesian product are produced dynamically (e.g., by computing $x \circ (A \ltimes x) \times (B \ltimes x)$, after $A \ltimes x$ and $B \ltimes x$ have essentially been computed). Thus a lot of time is saved from the avoidance of these joins. In addition, the performance improvement of LARA-J* over LARA-J increases with the number of Cartesian products that are avoided. For instance, the second query corresponds to the worst case for LARA-J, due to the unnecessary management of products `S`$\times$`T`, `S`$\times$`U`, `T`$\times$`U`, and `S`$\times$`T`$\times$`U`. On the other hand, the first query has products `R`$\times$`T` and `S`$\times$`U` only, with not as bad effect on the performance of LARA-J. When comparing LARA-J* to the HRJN plan, we observe that the plan of binary joins is more efficient in terms of time and memory requirements. This is expected since the intermediate results are fewer compared to LARA-J*, which materializes partial join results for more input combinations (i.e., for each connected subgraph).[10] On the other hand, LARA-J* incurs minimal accesses to the joined inputs with the savings being more significant in queries with more join edges. In summary, LARA-J* accesses (significantly) fewer tuples than plans of HRJN operators, while being much faster computationally than LARA-J.

7.5 Experiments with various aggregate functions

In [Mamoulis et al. 2006] we have compared NRA with versions of LARA for top-k `min` queries. Here, we compare LARA and NRA for the top-k queries with complex aggregate functions that are combinations of `min`, `max` and `sum`. We used six random synthetic datasets (D_1 to D_6) of $n = 50000$. Figure 23 shows the relative performance of the two methods for five queries (all with $k = 20$) that combine some of the datasets. As expected, applications of the `min` function are slower than those of `sum`, which in turn take more time than those of `max` for both LARA and NRA. Note that LARA performs better than NRA in terms of computations in all cases and the results are consistent with previous experiments with uniform data. The difference is due to the reasons we explained before for

[10]Note that the execution times of all algorithms are roughly proportional to the number of tuples that must be managed in memory.

Query	Function	HRJN	LARA-J	LARA-J*
chain	SELECT R.id, S.id, T.id FROM R, S, T, U WHERE R.a=S.a AND S.b=T.b AND T.c=U.c ORDER BY R.score+S.score+T.score+U.score STOP after k	CPU: 18msec accesses: 534 mem: 1311	CPU: 390msec accesses: 483 mem: 83356	CPU: 31msec accesses: 483 mem: 1030
star	SELECT R.id, S.id, T.id, U.id FROM R, S, T, U WHERE R.a=S.a AND R.b=T.b AND R.c=U.c ORDER BY R.score+S.score+T.score+U.score STOP after k	CPU: 18msec accesses: 560 mem: 1399	CPU: 7578msec accesses: 523 mem: 2081629	CPU: 31msec accesses: 523 mem: 1879
loop	SELECT R.id, S.id, T.id, U.id FROM R, S, T, U WHERE R.a=S.a AND S.b=T.b AND T.c=U.c AND U.b=R.b ORDER BY R.score+S.score+T.score+U.score STOP after k	CPU: 122msec accesses: 3484 mem: 18666	CPU: 2484msec accesses: 1636 mem: 372281	CPU: 328msec accesses: 1636 mem: 37787
other	SELECT R.id, S.id, T.id, U.id FROM R, S, T, U WHERE R.a=S.a AND S.b=T.b AND R.c=T.c AND T.a=U.a ORDER BY R.score+S.score+T.score+U.score STOP after k	CPU: 122msec accesses: 2983 mem: 15014	CPU: 5031msec accesses: 1562 mem: 942611	CPU: 219msec accesses: 1562 mem: 22609

Fig. 22. Top-k join with arbitrary join graphs (n = 50K, k = 20)

Query ID	Function	number of accesses		time in seconds	
		NRA	LARA	NRA	LARA
1	min(D_1,max(D_2,D_3,D_4))	2723	2639	5.469	0.016
2	min(sum(D_1,D_2),sum(D_3,D_4))	37656	37548	116.796	0.516
3	max(sum(D_1,D_2),sum(D_3,D_4))	3971	3968	1.875	0.047
4	sum(min(D_1,D_2),max(D_3,D_4))	15627	15626	23.796	0.250
5	sum(max(D_1,D_2),max(D_3,D_4,D_5))	3179	3175	1.250	0.062
6	$0.5 \cdot D_1 + D_2 + D_3$	27338	19589	16.516	0.110
7	$0.2 \cdot D_1 + D_2 + D_3$	40042	23741	18.843	0.110
8	$0.2 \cdot D_1 + 0.2 \cdot D_2 + D_3$	31208	24248	19.515	0.078
9	$0.2 \cdot D_1 + D_2 + D_3 + D_4$	107489	61560	67.797	0.281

Fig. 23. Comparison of LARA and NRA in complex aggregate (n = 50K, k = 20)

simpler queries. For easier queries (e.g., query 5) the difference not large, but it increases with the query complexity (e.g., see query 2). In terms of accesses, the difference is marginal due to the uniformity of the data.

Nevertheless, LARA has significant access cost saving over NRA for weighted queries (i.e., queries 6–9). As explained in Section 5.3.2, during the shrinking phase of LARA, most of the (unpruned) candidates have been accessed from sources with high weights, leading to early drying up of such sources. NRA does not dry up sources, therefore its access cost is significantly higher.

7.6 Experiments for top-k OLAP queries and browsing through top-k sets

We proceed to investigate the performance of the algorithms for top-k cube queries, as well as roll-up and drill-down queries.

Figure 24 shows the performance of the algorithms for top-k cube queries, on the FC dataset. Due to the drying up effect, LARA-CUBE performs 11% to 36% fewer accesses than NRA-CUBE. In addition, LARA-CUBE is at least 3 orders of magnitude faster than NRA-CUBE. The efficiency of LARA-CUBE renders it practical for top-k cube queries in real applications.

Figure 25 shows the performance of the algorithms for top-k roll-up/drill-down

attributes of FC	number of accesses		time in seconds	
	NRA-CUBE	LARA-CUBE	NRA-CUBE	LARA-CUBE
{*hh, hr, sa*}	1053485	841545	17312	3.1
{*hh, vh, hr, sn*}	2106736	1352031	55359	10.5
{*hh, hr, hf, vh, sp*}	1694683	1153573	50234	22.1
{*hh, hr, hf, vh, sa, sp*}	3219426	2886574	140922	132.7
{*hh, hr, hf, vh, sa, sn, sp*}	3798457	3157553	167531	376.6

Fig. 24. Top-k cube queries on forest coverage data (γ =`sum`, $k = 20$)

old attributes	new attributes	number of accesses		time in seconds	
		LARA	LARA-R	LARA	LARA-R
{*hh, hr, sa*}	{*hh, hr, sa, hf*}	818045	109156	2.969	2.906
{*hh, hr, sa, hf*}	{*hh, hr, sa*}	639400	59456	1.969	0.391
{*hh, hr, sa, hf*}	{*hh, hr, sa, hf, vh*}	1489431	842260	6.907	8.219
{*hh, hr, sa, hf, vh*}	{*hh, hr, sa, hf*}	818045	2	2.969	0.016
{*hh, hr, sa, hf, vh*}	{*hh, hr, sa, hf, vh, sp*}	2323667	952870	13.859	8.501
{*hh, hr, sa, hf, vh, sp*}	{*hh, hr, sa, hf, vh*}	1489431	60891	6.907	0.344

Fig. 25. Top-k roll-up/drill-down on forest coverage (γ =`sum`, $k = 20$)

queries, on the FC dataset. In Section 6.4, we propose LARA-R; incremental versions of LARA that accelerate processing of the new query by *reusing* the candidate set and data sources of the old query. We compare LARA-R with running LARA on the new query from scratch. Since LARA-R reuses data sources of the old query, its access cost on the new query is much lower than restarting LARA. In addition, LARA-R can be applied in problem settings where the data can be read only once. On the other hand, the CPU time of LARA-R is not necessarily smaller than LARA (see for example the third query of Figure 25). Recall that for LARA-R to be applicable, no candidates should be pruned during processing the old query. Keeping a large candidate size compromises the computational efficiency of running the new query. Nevertheless LARA-R is faster than applying LARA from scratch in most cases. Especially for roll-up queries the cost savings are very high, because very few additional accesses are required to derive the results. If fewer inputs are involved, the aggregate object scores are more distinguishable from each other and fewer accesses overall are required to finalize the result (see also Figure 18). On the other hand, in drill-down queries, the appearance of additional inputs requires accesses more scores until the top-k set is finalized. Therefore, LARA-R is generally faster faster at rolling-up than at drilling-down.

8. 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. We also propose and evaluate techniques for disk-based management of candidates by LARA, at large top-k problems.

By a theoretical analysis, we show that the expected per-access computational cost of LARA is $O(\log k + 2^m)$, where m is the number of aggregated inputs, which is much lower than the $O(|C|)$ cost of NRA ($|C|$ is the number of candidates). We also analyze the expected memory requirements for LARA for uniform data, as a parameter of m, n, and k. Our results can be used as a tool for predicting the allocated memory for NRA top-k search operators.

We also studied the application of LARA for queries with various aggregate functions, including weighted combinations of simple monotone functions. A case of top-k search that we studied in more depth are queries which rank (multiway) join results. For this class of problems, we proposed an effective adaptation of LARA, called LARA-J, which we demonstrated to have higher output rate compared to evaluation trees of binary HRJN operators [Ilyas et al. 2003]. To deal with the potentially high CPU cost and memory requirements of LARA-J, we proposed an optimized version of it (LARA-J*), which avoids the computation of Cartesian products derived from sparse query graphs and the join of any incoming tuple with them. Finally, we define the top-k cube query and browsing operations between top-k cuboids. For these queries we suggest effective adaptations of LARA and experimentally evaluate their performance. Summing up, this paper presents a set of powerful techniques for top-k search, in the common case where only sorted accesses to ranked inputs are allowed.

REFERENCES

Agrawal, R. and Wimmers, E. L. 2000. A framework for expressing and combining preferences. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 297–306.

Balke, W.-T. and Güntzer, U. 2004. Multi-objective query processing for database systems. In *Proceedings of the Very Large Databases Conference*. 936–947.

Beyer, K. S., Goldstein, J., Ramakrishnan, R., and Shaft, U. 1999. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory (ICDT)*. 217–235.

Börzsönyi, S., Kossmann, D., and Stocker, K. 2001. The skyline operator. In *Proceedings of the IEEE International Conference on Data Engineering*. 421–430.

Bruno, N., Chaudhuri, S., and Gravano, L. 2002. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems 27,* 2, 153–187.

Carey, M. J. and Kossmann, D. 1997. On saying "enough already!" in sql. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 219–230.

Chang, K. C.-C. and Hwang, S.-W. 2002. Minimal probing: supporting expensive predicates for top-k queries. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 346–357.

Chang, Y.-C., Bergman, L. D., Castelli, V., Li, C.-S., Lo, M.-L., and Smith, J. R. 2000. The onion technique: Indexing for linear optimization queries. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 391–402.

de Vries, A. P., Mamoulis, N., Nes, N., and Kersten, M. L. 2002. Efficient k-nn search on vertically decomposed data. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 322–333.

Fagin, R. 1999. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences 58,* 1, 83–99.

FAGIN, R. 2002. Combining fuzzy information: an overview. *SIGMOD Record 31,* 2, 109–118.

FAGIN, R., KUMAR, R., AND SIVAKUMAR, D. 2003. Efficient similarity search and classification via rank aggregation. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 301–312.

FAGIN, R., LOTEM, A., AND NAOR, M. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 102–113.

GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery 1,* 1, 29–53.

GÜNTZER, U., BALKE, W.-T., AND KIESSLING, W. 2000. Optimizing multi-feature queries for image databases. In *Proceedings of the Very Large Databases Conference*. 419–428.

GÜNTZER, U., BALKE, W.-T., AND KIESSLING, W. 2001. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the IEEE Int'l Conf. on Information Technology (ITCC)*. 622–628.

HRISTIDIS, V. AND PAPAKONSTANTINOU, Y. 2004. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal 13,* 1, 49–70.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2002. Joining ranked inputs in practice. In *Proceedings of the Very Large Databases Conference*. 950–961.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2003. Supporting top-k join queries in relational databases. In *Proceedings of the Very Large Databases Conference*. 754–765.

ILYAS, I. F., SHAH, R., AREF, W. G., VITTER, J. S., AND ELMAGARMID, A. K. 2004. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 203–214.

KIESSLING, W. 2002. Foundations of preferences in database systems. In *Proceedings of the Very Large Databases Conference*. 311–322.

MAMOULIS, N., CHENG, K. H., YIU, M. L., AND CHEUNG, D. W. 2006. Efficient aggregation of ranked inputs. In *Proceedings of the IEEE International Conference on Data Engineering.*

MARIAN, A., BRUNO, N., AND GRAVANO, L. 2004. Evaluating top- queries over web-accessible databases. *ACM Transactions on Database Systems 29,* 2, 319–362.

MOURATIDIS, K., BAKIRAS, S., AND PAPADIAS, D. 2006. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 635–646.

NATSEV, A., CHANG, Y.-C., SMITH, J. R., LI, C.-S., AND VITTER, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proceedings of the Very Large Databases Conference*. 281–290.

NEPAL, S. AND RAMAKRISHNA, M. V. 1999. Query processing issues in image (multimedia) databases. In *Proceedings of the IEEE International Conference on Data Engineering*. 22–29.

ORTEGA, M., RUI, Y., CHAKRABARTI, K., MEHROTRA, S., AND HUANG, T. 1997. Supporting similarity queries in MARS. In *Proceedings of the ACM International Multimedia Conference*. 403–414.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. 71–79.

TAO, Y., HRISTIDIS, V., PAPADIAS, D., AND PAPAKONSTANTINOU, Y. 2007. Branch-and-bound processing of ranked queries. *Information Systems 32,* 3, 424–445.

THEOBALD, M., WEIKUM, G., AND SCHENKEL, R. 2004. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Very Large Databases Conference*. 648–659.

TSAPARAS, P., PALPANAS, T., KOTIDIS, Y., KOUDAS, N., AND SRIVASTAVA, D. 2003. Ranked join indices. In *Proceedings of the IEEE International Conference on Data Engineering*. 277–288.

YI, K., YU, H., YANG, J., XIA, G., AND CHEN, Y. 2003. Efficient maintenance of materialized top-k views. In *Proceedings of the IEEE International Conference on Data Engineering*. 189–200.