Evaluating Top-k Queries over Web-Accessible Databases

Luis Gravano Amélie Marian Nicolas Bruno

Computer Science Department

Columbia University

{gravano,amelie,nicolas}@cs.columbia.edu

Abstract

A query to a web search engine usually consists of a list of keywords, to which the search engine responds with the best or "top" k pages for the query. This top-k query model is prevalent over multimedia collections in general, but also over plain relational data for certain applications. For example, consider a relation with information on available restaurants, including their location, price range for one diner, and overall food rating. A user who queries such a relation might simply specify the user's location and target price range, and expect in return the best 10 restaurants in terms of some combination of proximity to the user, closeness of match to the target price range, and overall food rating. Processing such top-k queries efficiently is challenging for a number of reasons. One critical such reason is that, in many web applications, the relation attributes might not be available other than through external web-accessible form interfaces, which we will have to query repeatedly for a potentially large set of candidate objects. In this paper, we study how to process top-k queries efficiently in this setting, where the attributes for which users specify target values might be handled by external, autonomous sources with a variety of access interfaces. We present several new algorithms for processing such queries, and adapt existing techniques to our scenario as well. We also study the execution time of our algorithms analytically and present experimental results using both synthetic and real web-accessible data. ***Index Terms:*** *Top-k query processing, query optimization, web databases.*

1 Introduction

A query to a web search engine usually consists of a list of keywords, to which the search engine responds with the best or "top" k pages for the query. This *top-k query model* is prevalent over multimedia collections in general, but also over plain relational data for certain applications where users do not expect exact answers to their queries, but instead a rank of the objects that best match a specification of target attribute values. Additionally, some applications require accessing data that resides at or is provided by remote, autonomous sources that exhibit a variety of access interfaces, which further complicates query processing.

Top-k queries arise naturally in applications where users have relatively flexible preferences or specifications for certain attributes, and can tolerate (or even expect) fuzzy matches for their queries. A top-k query in this context is then simply an assignment of target values to the attributes of a relation. To answer a top-k query, a database system identifies the objects that best match the user specification, using a given scoring function.

Example 1: *Consider a relation with information about restaurants in the New York City area. Each tuple (or object) in this relation has a number of attributes, including Address, Rating, and Price, which indicate, respectively, the restaurant's location, the overall food rating for the restaurant represented by a grade between 1 and 30, and the average price for a diner. A user who lives at 2590 Broadway and is interested in spending around $25 for a top-quality restaurant might then ask a top-10 query {Address="2590 Broadway", Price=$25, Rating=30}. The result to this query is a list of the 10 restaurants that match the user's specification the closest, for some definition of proximity.* ∎

Processing top-k queries efficiently is challenging for a number of reasons. One critical such reason is that, in many web applications, the relation attributes might not be available other than through external web-accessible form interfaces. For instance, in our example above the *Rating* attribute might be available through the Zagat-Review web site [1], which, given an individual restaurant name, returns its food rating as a number between 1 and 30 (*random access*). This site might also return a list of all restaurants ordered by their food rating (*sorted access*). Similarly, the *Price* attribute might be available through the New York Times's NYT-Review web site [2]. Finally, the scoring associated with the *Address* attribute might be handled by the MapQuest web site [3], which returns the distance (in miles) between the restaurant and the user addresses.

To process a top-k query over web-accessible databases, we then have to interact with sources that export different interfaces and access capabilities. In our restaurant example, a possible query processing strategy is to start with the Zagat-Review source, which supports sorted access, to identify a set of candidate restaurants to explore further. This source returns a rank of restaurants in decreasing order of food rating. To compute the final score for each restaurant and identify the top-10 matches for our query, we then obtain the proximity between each restaurant and the user-specified address by querying MapQuest, and check the average dinner price for each restaurant individually at the NYT-Review source. Hence, we interact with three autonomous sources and repeatedly query them for a potentially large set of candidate restaurants.

Fagin et al. [10] have presented query processing algorithms for top-k queries for the case where all intervening sources support sorted access (plus perhaps random access as well). These algorithms are not designed for sources that only support random access (e.g., the MapQuest site), which abound on the web [4]. In this paper,

[1] `http://www.zagat.com`
[2] `http://www.nytoday.com`
[3] `http://www.mapquest.com`
[4] Recently, in an expanded version of their paper, Fagin et al. introduced a variation of their algorithm for random-access sources. See Section 2.

we present novel processing strategies for top-k queries over sources that support just random access, or both random and sorted access. We also develop improvements of Fagin et al.'s algorithms, and compare these techniques experimentally using synthetic and real web-accessible data sets.

The rest of the paper is structured as follows. Section 2 defines our query and data models, notation and terminology that we use in Section 3 to present our new techniques and our adaptations of Fagin et al.'s algorithms. In Section 4 we introduce the data structures that we use to speed up the local processing of our techniques, and in Section 5 we report a time and space complexity analysis of the algorithms. We evaluate the different strategies experimentally in Section 7 using the data sets and metrics in Section 6. In Section 8 we discuss generalizations of our data model. Finally, in Section 9 we review relevant work.

2 Data and Query Models

In traditional relational systems, query results consist of an unordered set of tuples. In contrast, the answer to a *top-k query* is an *ordered* set of tuples, where the ordering is based on how close each tuple matches the given query. Furthermore, the answer to a top-k query does not include all tuples that "match" the query, but rather only the best k such tuples. In this section we define our data and query models in detail.

Consider a relation R with attributes $A_0, A_1, \ldots, A_n$, plus perhaps some other attributes not mentioned in our queries. A top-k query over relation R simply specifies target values for the attributes A_i. Therefore, a top-k query is an assignment of values $\{A_0 = q_0, A_1 = q_1, \ldots, A_n = q_n\}$ to the attributes of interest. Note that some attributes might always have the same "default" target value in every query. For example, it is reasonable to assume that the *Rating* attribute in Example 1 above might always have an associated query value of 30. (It is unclear why a user would insist on a lesser-quality restaurant, given the target price specification.) In such cases, we simply omit these attributes from the query, and assume default values for them.

Consider $q = \{A_0 = q_0, A_1 = q_1, \ldots, A_n = q_n\}$, a top-$k$ query over a relation R. The score that each tuple (or *object*) t in R receives for q is a function of t's score for each individual attribute A_i with target value q_i. Specifically, each attribute A_i has an associated *scoring function* $Score_{A_i}$ that assigns a proximity score to q_i and t_i, where t_i denotes the value of object t for attribute A_i. To combine these individual attribute scores into a final score for each object, each attribute A_i has an associated weight w_i indicating its relative importance in the query. Then, the final score for object t is defined as a weighted sum of the individual scores [5]:

$$Score(q,t) = ScoreComb(s_0, s_1, \ldots, s_n) = \sum_{i=0}^{n} w_i \cdot s_i$$

where $s_i = Score_{A_i}(q_i, t_i)$. The result of a top-$k$ query is the ranked list of the k objects with highest *Score* value, where we break ties arbitrarily.

[5] Our model and associated algorithms can be adapted to handle other scoring functions (e.g., min), which we believe are less meaningful than weighted sums for the applications that we consider.

Example 1: (cont.) *We can define the scoring function for the Address attribute of a query and an object as the inverse of the distance (say, in miles) between the two addresses. Similarly, the scoring function for the Price attribute might be a function of the difference between the target price and the object's price, perhaps "penalizing" restaurants that exceed the target price more than restaurants that are below it. The scoring function for the Rating attribute might simply be the object's value for this attribute. If price and quality are more important to a given user than the location of the restaurant, then the query might assign, say, a* 0.2 *weight to attribute Address, and a* 0.4 *weight to attributes Price and Rating.* ∎

Recent techniques to evaluate top-k queries over traditional relational DBMSs [1, 8] assume that all attributes of every object are readily available to the query processor. However, in many applications some attributes might not be available "locally," but rather will have to be obtained from an external web-accessible source instead. For instance, the *Price* attribute in our example is provided by the NYT-Review web site and can only be accessed by querying this site's web interface [6].

This paper focuses on the efficient evaluation of top-k queries over a (distributed) "relation" whose attributes are handled and provided by autonomous sources accessible over the web with a variety of interfaces. Specifically, we distinguish between three types of sources based on their access interface:

Definition 1: [**Source Types**] *Consider an attribute A_i with target value q_i in a top-k query q. Assume further that A_i is handled by a source S. We say that S is an* ***S-Source*** *if, given q_i, we can obtain from S a list of objects sorted in descending order of $Score_{A_i}$ by (repeated) invocation of a* $\texttt{getNext}_S(q_i)$ *probe interface. Alternatively, assume that A_i is handled by a source R that only returns scoring information when prompted about individual objects. In this case, we say that R is an* ***R-Source****. R provides random access on A_i through a* $\texttt{getScore}_R(q_i, t)$ *probe interface, where t is a set of attribute values that identify an object in question. (As a small variation, sometimes an R-Source will return the actual attribute A_i value for an object, rather than its associated score.) Finally, we say that a source that provides both sorted and random access is an* ***SR-Source****.*

Example 1: (cont.) *In our running example, attribute Rating is associated with the Zagat-Review web site. This site provides both a list of restaurants sorted by their rating (sorted access), and the rating of a specific restaurant given its name (random access). Hence, Zagat-Review is an SR-Source. In contrast, Address is handled by the MapQuest web site, which returns the distance between the restaurant address and the user-specified address. Hence, MapQuest is an R-Source.* ∎

To define query processing strategies for top-k queries involving the three source types above, we need to consider the cost that accessing such sources entails:

[6]Of course, in some cases we might be able to download all this remote information and cache it locally with the query processor. However, this will not be possible for legal or technical reasons for some other sources, or might lead to highly inaccurate or outdated information.

Definition 2: [**Access Cost**] *Consider a source R that provides a random-access interface, and a top-k query. We refer to the **average time** that it takes R to return the score for a given object as **$tR(R)$**. (tR stands for "random-access time.") Similarly, consider a source S that provides a sorted access interface. We refer to the average time that it takes S to return the top object for the query as **$tS(S)$**. (tS stands for "sorted-access time.") We make the simplifying assumption that successive invocations of the* `getNext` *interface also take time $tS(S)$ on average.*

Fagin et al. [10] presented "instance optimal" query processing algorithms over sources that are either of type *SR-Source* (TA algorithm) or of type *S-Source* (NRA algorithm). In an expanded version of [10], Fagin et al. introduced the TA_z algorithm, a variation of *TA* that handles both *SR-Sources* and *R-Sources*. These algorithms completely "process" one object before moving to another object. As we will see, by interleaving random-access probes on different objects, the query processing time can be dramatically reduced. In the remainder of this paper, we will present efficient top-k query processing techniques that take advantage of the interleaving of probes on objects, as well as adaptations of existing algorithms.

3 Evaluating Top-k Queries

In this section we present strategies for evaluating top-k queries, as defined in Section 2. Specifically, in Section 3.1 we present a naive but expensive approach to evaluate top-k queries. Then, in Section 3.2 we introduce our novel strategies. Finally, in Section 3.3 we adapt existing techniques designed for similar problems to our framework.

We make a number of simplifying assumptions in our presentation. Specifically, we assume that the scoring function for all attributes return scores between 0 and 1, with 1 denoting a perfect match. Also, we assume that exactly one *S-Source*, denoted S and associated with attribute A_0, and multiple *R-Sources*, denoted $R_1, \ldots, R_n$ and associated with attributes $A_1, \ldots, A_n$, are available. (The *S-Source* S could in fact be of type *SR-Source*. In such a case, we will ignore its random-access capabilities in our discussion.) In addition, we assume that only one source is accessed at a time, so all probes are sequential during query processing. We discuss relaxations of this source model and present algorithms over one or more *SR-Sources* and arbitrarily many *R-Sources* in Section 8.

Following Fagin et al. [9, 10], we do not allow our algorithms to rely on "wild guesses": thus a random access cannot zoom in on a previously unseen object, i.e., on an object that has not been previously retrieved under sorted access from a source. Therefore, an object will have to be retrieved from the *S-Source* before being probed on any *R-Source*. Since we have exactly one *S-Source* S available, objects in S are then the only candidates to appear in the answer to a top-k query. We refer to this set of candidate objects as *Objects*(S). Besides, we assume that all *R-Sources* $R_1, \ldots, R_n$ "know about" all objects in *Objects*(S). In other words, given a query q and an object $t \in \mathit{Objects}(S)$, we can probe R_i and obtain the score $\mathit{Score}_{A_i}(q_i, t_i)$ corresponding to q and t for attribute A_i, for all $i = 1, \ldots, n$. Of course, this is a simplifying assumption that is likely not to hold in practice, where each *R-Source* might be autonomous and not coordinated in any way with the other sources. For instance, in our running example the NYT-Review site might not have reviewed a specific restaurant, and hence it will not be able to return a score for the *Price* attribute for such a restaurant. In this case, we use a default value for $\mathit{Score}_{A_i}(q_i, t_i)$.

3.1 A Naive Strategy

A simple technique to evaluate a top-k query q consists of retrieving all partial scores for each object in $\mathit{Objects}(S)$, calculating the corresponding combined scores, and finally returning k objects with the highest scores. This simple procedure returns a correct answer to the given top-k query. However, we need to retrieve all scores for each object in $\mathit{Objects}(S)$. This can be unnecessarily expensive, especially since many scores are not needed to produce the final answer for the query, as we will see. Using Definition 2, this strategy takes time $|\mathit{Objects}(S)| \cdot (tS(S) + \sum_{i=1}^{n} tR(R_i))$.

3.2 Our Proposed Strategies

In this section we present novel strategies to evaluate top-k queries over one *S-Source* and multiple *R-Sources*. Our techniques lead to efficient executions by explicitly modeling the time of random probes to *R-Sources*. Unlike the naive strategy of Section 3.1, our algorithms choose *both* the best object *and* the best attribute on which to probe next at each step. In fact, we will in general not probe all attributes for each object under consideration, but only those needed to identify a top-k answer for a query.

Consider a top-k query q and an intermediate step in some execution of the query. Suppose that an object t has been retrieved from *S-Source* S and that we have already probed some subset of *R-Sources* $R' \subseteq \{R_1, \ldots, R_n\}$ for this object. Let $s_i = \mathit{Score}_{A_i}(q_i, t_i)$ if $R_i \in R'$. (Otherwise, s_i is undefined.) Then, an *upper bound for the score of object* t, denoted $U(t)$, is the maximum possible score that object t can get, consistent with the information from the probes that we have already performed. $U(t)$ is then the score that t would get for q if t had the maximum score of 1 for every attribute in the query that has not yet been processed for t: $U(t) = \mathit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n)$, where $\hat{s}_i = s_i$ if $R_i \in R'$, and $\hat{s}_i = 1$ otherwise. If object t has not been retrieved from S yet, then we define $U(t) = \mathit{ScoreComb}(s_\ell, 1, \ldots, 1)$, where s_ℓ is the Score_{A_0} score for the last object retrieved from S, or 1 if no object has been retrieved yet. (t's score for A_0 cannot be larger than s_ℓ, since *S-Source* S returns objects in descending order of Score_{A_0}.)

Similarly, a *lower bound for the score of an object* t already retrieved from S, denoted $L(t)$, is the minimum possible score that object t can get for q: $L(t) = \mathit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n)$, where $\hat{s}_i = s_i$ if $R_i \in R'$, and $\hat{s}_i = 0$ otherwise. If object t has not been retrieved from S yet, then we define $L(t) = 0$.

Finally, the *expected score for an object t* already retrieved from S, denoted $E(t)$, is obtained by assuming that the score for each attribute that has not yet been probed is some expected partial score $e(A_i)$: $E(t) = \mathit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n)$, where $\hat{s}_i = s_i$ if $R_i \in R'$, and $\hat{s}_i = e(A_i)$ otherwise. If object t has not been retrieved from S yet, then we define $E(t) = \mathit{ScoreComb}(e(A_0), e(A_1), \ldots, e(A_n))$. In the absence of additional information we set the expected partial score $e(A_i)$ to 0.5 for $i = 1, ..., n$, while $e(A_0) = \frac{s_\ell}{2}$, where s_ℓ is the Score_{A_0} score for the last object retrieved from S, or 1 if no object has been retrieved yet [7]. ($\mathit{Score}_{A_0}(q_0, t_0)$ can range between 0 and s_ℓ.)

In Section 3.2.1 we define what constitutes an optimal query processing strategy in our framework. In Section 3.2.2 we describe one new strategy, *Upper*, which can be seen as mimicking the optimal solution when no complete information is available.

3.2.1 The Optimal Strategy

Given a top-k query q, the *Optimal* strategy for evaluating q is the most efficient sequence of `getNext` and `getScore` calls that produce top-k objects for the query along with their scores. Furthermore, such an optimal strategy must also provide enough evidence (in the form of at least partial scores for additional objects) to demonstrate that the returned objects are indeed a correct answer for the top-k query. In this section we show one such optimal strategy, built assuming complete knowledge of the object scores. Of course, this is not a realistic query processing technique, but it provides a useful lower bound on the time of any processing strategy without "wild guesses." Additionally, the optimal strategy provides useful insight that we exploit to define an efficient algorithm in the next section.

As a first step towards our optimal strategy, consider the following property of any top-k query processing algorithm:

Property 1: *Consider a top-k query q and suppose that, at some point in time, we have retrieved a set of objects T from S-Source S and probed some of the R-Sources for these objects. Assume further that the score upper bound $U(t)$ for an object $t \in \mathrm{Objects}(S)$ is strictly lower than the score lower bound $L(t_i)$ for k different objects $t_1, \ldots, t_k \in T$. Then t is guaranteed not to be one of the top-k objects for q.*

Using this property, we can view an optimal processing strategy as (a) computing the final scores for k top objects for a given query, which are needed in the answer, while (b) probing the fewest and least expensive attributes on the remaining objects so that their score upper bound for the query is no higher than the scores of the top-k objects. (We can safely discard objects with upper bound matching the lowest top-k object score since we break ties arbitrarily.) This way, an optimal strategy identifies and scores the top objects, while providing enough evidence that the rest of the objects have been safely discarded.

[7] Alternative techniques for estimating expected partial scores include sampling and exploiting attribute-score correlation if known.

Algorithm Optimal (Input: top-k query q)

1. Choose a set of k objects, $Answer_k$, such that $Answer_k$ is a solution to the top-k query q [8]. (*Optimal* assumes complete knowledge of all object scores.)
2. Let $score_k = \min_{t \in Answer_k}\{Score(q,t)\}$.
3. Repeat
   (a) Get the best unretrieved object t for Attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   (b) Set $U_{unseen} = U(t)$ (no unretrieved object from S can have a score larger than U_{unseen}).
   (c) If object t is one of the $Answer_k$ objects, probe all *R-Sources* to compute $Score(q,t)$.
   Otherwise, probe a subset $R' \subseteq \{R_1, \ldots, R_n\}$ for t such that:
      - After probing every $R_i \in R'$, it holds that $U(t) \leq score_k$.
      - The time $\sum_{R_i \in R'} tR(R_i)$ is minimal among the subsets of $\{R_1, \ldots, R_n\}$ with the property above.

   Until $U_{unseen} \leq score_k$ and we have retrieved all objects in $Answer_k$.

The *Optimal* algorithm is only of theoretical interest and cannot be implemented, since it requires complete knowledge about the scores of the objects, which is precisely what we are trying to obtain to evaluate top-k queries.

3.2.2 The Upper Strategy

We now present a novel top-k query processing strategy that we call *Upper*. This strategy mimics the *Optimal* algorithm by choosing probes that would have the best chance to be in the *Optimal* solution. However, unlike *Optimal*, *Upper* does not assume any "magic" a-priori information on object scores. Instead, at each step *Upper* selects an object-source pair to probe next based on *expected* object scores. This chosen pair is the one that would hopefully have been in the *optimal* set of probes.

We can observe an interesting property:

Property 2: *Consider a top-k query q and suppose that at some point in time we have retrieved some objects from S-Source S and probed some of the R-Sources for these objects. Suppose that an object $t \in \text{Objects}(S)$ has a score upper bound $U(t)$ strictly higher than that of every other object (i.e., $U(t) > U(t')\ \forall t' \neq t \in \text{Objects}(S)$). Then, at least one probe will have to be done on t before the answer to q is reached:*

- *If t is one actual top-k object, then we need to probe all of its attributes to return its final score for q.*
- *If t is not one of the actual top-k objects, t requires further probes to decrease its score upper bound $U(t)$ since $U(t)$ is higher than the score of each top-k object.*

This property is illustrated in Figure 1 for a top-3 query. In this figure, the possible range of scores for each object is represented by a segment, and objects are sorted by their expected score. From Property 1, objects

[8] In the presence of score ties, to ensure optimality, this step picks the objects that would be the most expensive to discard in Step 3(c).

whose upper bound is lower than the lower bound of k other objects cannot be in the final answer. (Those objects are marked with a dashed segment in Figure 1.) From Property 2, the object with the highest score upper bound, noted **U** in the figure, will have to be probed before a solution is reached: either **U** is one of the top-3 objects for the query and its final score needs to be returned, or its score upper bound will have to be lowered through further probes so that we can safely discard the object.

We exploit Properties 1 and 2 and the general structure of the *Optimal* algorithm to define our *Upper* algorithm:

Algorithm Upper (Input: top-k query q)

1. Initialize $U_{unseen} = 1$, $Candidates = \emptyset$, and $returned = 0$.
2. While $(returned < k)$
   (a) If $Candidates \neq \emptyset$, pick $t_H \in Candidates$ such that $U(t_H) = \max_{t \in Candidates} U(t)$.
   Else t_H is undefined.
   (b) If t_H is undefined or $U(t_H) < U_{unseen}$ (unseen objects might have larger scores than all candidates):
   - Get the best unretrieved object t for attribute A_0 from S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   - Update $U_{unseen} = U(t)$ and insert t into $Candidates$.

   Else If t_H is completely probed (t_H is one of the top-k objects):
   - Return t_H with its score; remove t_H from $Candidates$.
   - $returned = returned + 1$.

   Else:
   - $Ri \leftarrow$ *SelectBestSource*(t_H, *Candidates*).
   - Probe source R_i on object t_H: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t_H)$.

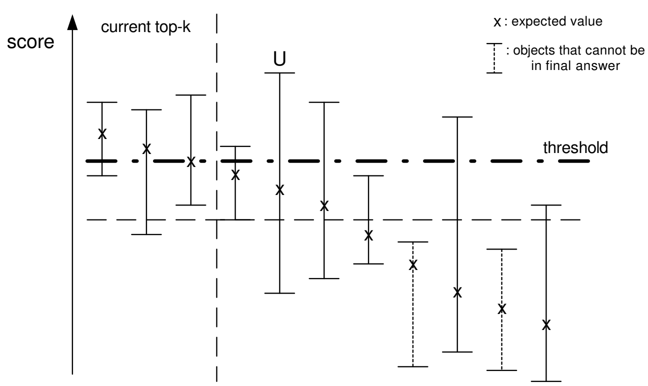


Figure 1: Snapshot of the execution of the *Upper* strategy.

At any point in time, if the final score of the object with the highest upper bound is known, then this is the best object in the current set. No other object can have a higher score and we can safely return this object as one of the top-k objects for the query. As a corollary, *Upper* can return results as they are produced, rather than having to wait for all top-k results to be known before producing the final answer.

We now discuss how we select the best source to probe for an object t (function *SelectBestSource* in the *Upper* algorithm). As in *Optimal*, we concentrate on (a) computing the final score of the top-k objects, and (b) for all other objects, decreasing their upper bound to not exceed the scores of the top-k objects. However, unlike *Optimal*, *Upper* does not know the actual object scores a priori and must rely on expected scores to make its choices. In particular, we will *estimate* the value $score_k$ used in the *Optimal* algorithm (i.e., the k-th top score) using $score'_k$, the k-th largest *expected* score in the set of candidates. Below, we introduce three versions of *Upper* that differ in their underlying variant of the *SelectBestSource* function.

Upper-Greedy. A simple and efficient implementation of *SelectBestSource* selects the next source to probe for an object t as the *R-Source* with the highest expected "impact" in the smallest amount of time. Specifically, the expected decrease of $U(t)$ after probing source R_i is given by $\delta_i = w_i \cdot (1 - e(A_i))$, where w_i is the weight of attribute A_i in the query (Section 2), and $e(A_i)$ is the expected score for source R_i. The ratio $\delta_i/tR(R_i)$ is then a good indicator of the "efficiency" of source R_i: a large value of this ratio indicates that we can reduce the value of $U(t)$ by a large amount (δ_i) relative to the time that the associated probe insumes ($tR(R_i)$).

Sometimes we do not need large reductions δ_i to discard a borderline candidate object; in this case, a fast source R_j with a sufficiently large δ_j value might be preferred to a slower source R_i with a larger δ_i value. This observation motivates the following definition of the "goodness" $Rank(R_i)$ of a source R_i for a candidate object t. Before introducing this definition, we consider two scenarios:

- *Case 1:* $E(t) < score'_k$. In this case, t is not expected to be one of the top-k objects. Furthermore, $\Delta = U(t) - score'_k$ is the amount by which we need to decrease $U(t)$ to "prove" that t is not one of the top-k answers. (As discussed above, $score'_k$ is the k^{th} largest expected score in the set of candidate objects.) In other words, it does not really matter how large the decrease of $U(t)$ is beyond Δ when choosing the best probe for t. However, using the previous ratio, we might choose some source R_i with a very large value of δ_i, $\delta_i \gg \Delta$, but not a particularly fast response time $tR(R_i)$. In such a case, a better choice would be some faster source R_j with $tR(R_j) < tR(R_i)$ that is expected to decrease $U(t)$ by at least Δ as well, even if $\delta_j/tR(R_j) < \delta_i/tR(R_i)$.
- *Case 2:* $E(t) \geq score'_k$. In this case, t is expected to be one of the top-k objects. Hence, t needs to be completely probed and then all probes are equally good for t.

We now define the *Upper-Greedy* variation of our *Upper* algorithm. *Upper-Greedy* considers all *R-Source* sources not yet probed for the current object t, and chooses a source with the highest value of rank:

$$Rank(R_i) = \frac{Min\{\Delta, \delta_i\}}{tR(R_i)}$$

We extensively tested this modified rank metric and found that it always resulted in faster executions than the simpler $\delta_i/tR(s_i)$ ratio did.

Upper-Filter. Although the heuristic used in the *Upper-Greedy* technique to pick sources for probing is efficient, it sometimes results in provably sub-optimal choices, as illustrated in the following example.

Example 2: *Consider an object t and two R-Sources R_1 and R_2, with access times $tR(R_1)$=1 and $tR(R_2)$=10, and query weights w_1=0.1 and w_2=0.9. Assume that $score'_k$=0.5 and $U(t) = 0.9$, so the amount by which we need to decrease t to prove it is not one of the top answers is $\Delta = 0.4$. If we assume that $e(A_1)$=$e(A_2)$=0.5, Upper-Greedy would choose source R_1 (with rank $\frac{Min\{0.4,0.05\}}{1} = 0.05$) over source R_2 (with rank $\frac{Min\{0.4,0.45\}}{10} = 0.045$). However, we know that we will need to eventually lower $U(t)$ below $score'_k$=0.5, and that R_1 can only decrease $U(t)$ by 0.1 to 0.8, since w_1=0.1. Therefore, in subsequent iterations, source R_2 would need to be probed anyway. In contrast, if we start with source R_2, we might decrease $U(t)$ below $score'_k = 0.5$ thus avoiding a probe to source R_1 for t.* ∎

The previous example shows that, for a particular object t, a source R_i can be "redundant" independently of its rank $Min\{\Delta, \delta_i\}/tR(R_i)$. Therefore, such a source should not be probed for t before the "non-redundant" sources. The set of redundant sources is not static, but rather depends on the execution state of the algorithm. (In the example above, if $score'_k = 0.89$, there are no redundant sources.) We now refine *Upper-Greedy* by first identifying the subset of non-redundant available sources: if an object t is expected to be in the final answer (i.e., $E(t) \geq score'_k$), we need to compute its final score, so all available sources for t are non-redundant and are kept as candidates, just as in *Upper-Greedy*. Otherwise, we identify redundant sources in the following way. Let $\Delta = U(t) - score'_k$ as above and let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be the set of available sources. We say that source R_i is redundant for object t at a given step of the probing process if:

1. $w_i < \Delta$ (i.e., source R_i by itself cannot decrease the value of $U(t)$ below $score'_k$), and
2. $\forall Y \subseteq \mathcal{R} - \{R_i\}$: If $w_i + \sum_{j:R_j \in Y} w_j \geq \Delta$ then $\sum_{j:R_j \in Y} w_j \geq \Delta$ (i.e., for every possible choice of sources $\{R_i\} \cup Y$ that can decrease $U(t)$ below $score'_k$, Y by itself can also do it).

By negating the predicate above, replacing the implication with the equivalent disjunction, and manipulating the resulting predicate, we obtain the following test to identify non-redundant sources: R_i is non-redundant if and only if $(w_i \geq \Delta) \vee (\exists Y \subseteq \mathcal{R} - \{R_i\} : \Delta - w_i \leq \sum_{j:R_j \in Y} w_j < \Delta)$. It is not difficult to prove that for any possible assignment of values to w_i and Δ, there is always at least one available non-redundant source. Therefore, after identifying the subset of non-redundant sources, *Upper-Filter* returns the *non-redundant source* with the maximum rank $\frac{Min\{\Delta,\delta_i\}}{tR(R_i)}$, just as *Upper-Greedy* does.

Upper-Subset. Finally, we report the alternative formulation for *SelectBestSource* that we presented in [2] [9]. Consider an object t. If t is expected to be in the final answer (i.e., $E(t) \geq score'_k$), we need to get its final

[9] The experimental results show that *Upper-Subset* is slightly worse than *Upper-Filter*. However, *Upper-Subset* is an interesting strategy in that it can be more easily adapted to handle a query execution scenario in which several probes can proceed in parallel.

score, so we consider all available sources just as *Upper-Greedy* does. Otherwise, we identify the fastest subset of sources not yet probed for t that is expected to decrease $U(t)$ to not exceed the value of $score'_k$. (Since $E(t) < score'_k$, we are guaranteed to find at least one such set of attributes.) In other words, we identify $R' \subseteq \{R_1, \ldots, R_n\}$ so that:

1. $U(t) \leq score'_k$ if each source $R_i \in R'$ were to return the expected score for t, and
2. The access time $\sum_{R_i \in R'} tR(R_i)$ is minimal among the subsets of $\{R_1, \ldots, R_n\}$ with the above property.

As with *Upper-Filter*, after we identify the subset R' above, we return the source $R_i \in R'$ with the maximum rank $\frac{Min\{\Delta, \delta_i\}}{tR(R_i)}$.

3.3 Existing Approaches

In Section 3.3.1 we adapt Fagin et al.'s TA algorithm [10] so that it also works over *R-Sources* [10], and in Section 3.3.2 we extend the resulting algorithm so that it also incorporates ideas from the literature on processing selection queries involving expensive predicates. As an important difference with our strategies of the previous section, the techniques below choose an object and probe *all* needed sources before moving to the next object. This "coarser" strategy can degrade the overall efficiency of the techniques, as shown in Section 7.

3.3.1 Fagin et al.'s Algorithms

Fagin et al. [10] presented the *TA* algorithm for processing top-k queries over *SR-Sources*:

Algorithm TA (Input: top-k query q)

1. Do sorted access in parallel to each source. As each object t is seen under sorted access in one source, do random accesses to the remaining sources and apply the *Score* function to find the final score of object t. If $Score(q, t)$ is one of the top-k scores seen so far, keep object t along with its score.
2. Define a threshold value as $ScoreComb(s_0, s_1, \ldots, s_n)$, where s_i is the last score seen in the *i-th* source. The threshold represents the highest possible score of any object that has not been seen so far in any source.
3. If the scores of the current top-k objects seen so far are greater than or equal to the threshold, return the top-k objects and stop. Otherwise, return to step 1.

Although this algorithm is not designed for *R-Sources*, we can adapt it in the following way. In step 1, we access the only *S-Source* S using sorted access. In step 2, we define the threshold value as $U(t)$, where t is

[10]Recently, Fagin et al. expanded their paper [10] to include a variation of *TA*, TA_z, that works over any number of *R-Sources* and *SR-Sources*. Our first adaptation of *TA* (i.e., *TA-Adapt*) is identical to TA_z for the one-*SR-Source* case.

the last object retrieved from S under sorted access. (The maximum possible score for any *R-Source* is always 1.) For each object retrieved from S, we probe all *R-Sources* to get its final score. For a model with a single *S-Source* S, the modified algorithm retrieves the objects in *Objects*(S) in order, one by one, and determines whether each object is in the final answer by probing the remaining *R-Sources*. The complete procedure, called ***TA-Adapt***, is described next.

Algorithm TA-Adapt (Input: top-k query q)

1. Repeat
   (a) Get the best unretrieved object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   (b) Update threshold $T = U(t)$.
   (c) For each *R-Source* R_i, retrieve score s_i for attribute A_i and object t via a random probe to R_i: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t)$.
   (d) Calculate t's final score for q: $score = ScoreComb(s_0, s_1, \ldots, s_n)$. If *score* is one of the top-k scores seen so far, keep object t along with its score.

   Until we have seen at least k objects and T is no larger than the scores of the current k top objects.
2. Return the top-k objects along with their score.

We can improve the algorithm above by interleaving the execution of steps (1-c) and (1-d) and adding a shortcut test condition. Given an object t, we calculate the value $U(t)$ after each random probe to an *R-Source* R_i, and we skip directly to after step (1-d) if the current object t is guaranteed not to be better than k top objects. That is, if $U(t)$ is no higher than the score of k objects, we can safely ignore t (Property 1) and continue with the next object. We call this algorithm ***TA-Opt***, and we present it below:

Algorithm TA-Opt (Input: top-k query q)

1. Repeat
   (a) Get the best unretrieved object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   (b) Update threshold $T = U(t)$.
   (c) For each *R-Source* R_i:
      i. Retrieve score s_i for attribute A_i and object t via a random probe to R_i: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t)$.
      ii. If $U(t)$ is less than or equal to the score of k objects, skip to (1-d).
   (d) If we probed t completely, calculate t's final score for q: $score = ScoreComb(s_0, s_1, \ldots, s_n)$. If t's score is one of the top-k scores seen so far, keep object t along with its score.

   Until we have seen at least k objects and T is no larger than the scores of the current k top objects.
2. Return the top-k objects along with their score.

3.3.2 Exploiting Techniques for Processing Selections with Expensive Predicates

Research on expensive-predicate query optimization [13, 15] has studied how to process selection queries of the form $p_1 \wedge \ldots \wedge p_n$, where each predicate p_i can be expensive to calculate. The key idea is to order the evaluation of predicates to minimize the expected execution time. The evaluation order is determined by the predicates'

Rank, defined as $Rank(p_i) = \frac{1-selectivity(p_i)}{cost-per-object(p_i)}$, where *selectivity*$(p_i)$ is the fraction of the objects that are estimated to satisfy p_i, and *cost-per-object*(p_i) is the average time to evaluate p_i over an object.

We can adapt this idea to our framework as follows. Let $R_1, \ldots, R_n$ be the *R-Sources*, with weights $w_1, \ldots, w_n$ in the *Score* function. Similarly to what we described in Section 3.2.2 for *Upper-Greedy*, if $e(A_i)$ is the expected score for source R_i, the expected decrease of $U(t)$ after probing source R_i is $\delta_i = w_i \cdot (1 - e(A_i))$. The analogous definition for Δ in this situation is $\Delta = U(t) - U(t_k)$, where t_k is the *k-th* top object seen so far. The magnitude of the decrease of $U(t)$ beyond Δ is unimportant since our shortcut condition is precisely $U(t) < U(t_k)$. We sort the *R-Sources* R_i in decreasing order of *Rank*, defined as: $Rank(R_i) = \frac{Min\{\Delta, \delta_i\}}{tR(R_i)}$. Thus, we favor fast sources that might have a large impact on the final score of an object, i.e., those sources that are likely to significantly change the value of $U(t)$ fast.

We combine this idea with our adaptation of the TA algorithm to define the ***TA-EP*** algorithm:

Algorithm TA-EP (Input: top-k query q)

1. Repeat
   (a) Get the best unretrieved object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   (b) Update threshold $T = U(t)$.
   (c) For each *R-Source* R_i in decreasing order of $Rank(R_i)$:
      i. Retrieve score s_i for attribute A_i and object t via a random probe to R_i: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t)$.
      ii. If $U(t)$ is less than or equal to the score of k objects, skip to (1-d).
   (d) If we probed t completely, calculate t's final score for q: $score = ScoreComb(s_0, s_1, \ldots, s_n)$. If t's score is one of the top-k scores seen so far, keep object t along with its score.

   Until we have seen at least k objects and T is no larger than the scores of the current k top objects.
2. Return the top-k objects along with their score.

In this section we presented novel strategies and adapted existing ones for processing top-k queries. We gave detailed descriptions for each algorithm, but not for the data structures needed to support these algorithms. As we discuss next, a careful choice of data structures can result in efficient executions of the different algorithms.

4 Supporting Data Structures

In this section we describe the data structures that we use to implement the algorithms of Section 3. In particular, in Section 4.1 we show how we represent objects and combine score information returned by the sources. Then, in Section 4.2 we present data structures to efficiently maintain ranked object sets (e.g., according to the score upper bounds of the objects).

4.1 Representing Objects

Our algorithms need to keep track of the objects retrieved and their partial score information. We maintain this information in a hash table indexed by the object ids. Figure 2 shows the representation of an arbitrary object. This representation consists of an object id and $n+1$ slots for the attribute scores returned by the different sources. A special value, denoted as $*$, is used when information about a particular source is not yet available. These $n+1$ values (along with the query weights) are sufficient to calculate the score of an object, or the lower and upper bounds of an object's score when some attribute scores are not available. However, for efficiency, we also incrementally maintain the lower and upper bounds as separate fields in the same object (shown as L and U in the figure). Whenever the lower and upper bounds coincide (i.e., $L = U$), we know that no additional probe is needed for this object and its final score is equal to the bounds. Finally, depending on the algorithm, each object is augmented with a small number of pointers. As we will see in the next section, these pointers help us to efficiently maintain the rank of each object in different ordered lists.

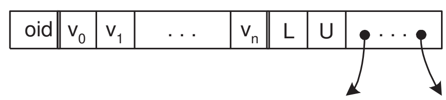


Figure 2: Representation of an object and its scores for a query.

4.2 Ranking Objects

During the execution of the algorithms of Section 3, each object can be part of multiple sorted lists. As an example, all variations of the *Upper* algorithm of Section 3.2.2 need to keep track of the object with the largest score upper bound (Step (2-a) in the algorithm). All variations of the *SelectBestSource* function also need to identify the object with the *k-th* highest expected score. We implement each sorted list using heap-based priority queues, which provide constant-time access to the first ranked element, and logarithmic-time insertions and deletions. Later, we will show how to modify these standard priority queues to extract in constant time the *k-th* ranked object in the list still with logarithmic-time insertions and deletions.

Standard Priority Queues: Each node in a priority queue consists of a pointer to the object it represents (to avoid duplicate information) and the ranking for the priority queue. Figure 3 shows the representation of object o_1 and its corresponding node in a priority queue sorted by score upper bounds. Three out of the four attribute scores for o_1 are known. Assuming that all weights in the query are equal to one, the lower and upper bounds for object o_1 are 1.6 and 2.6, respectively. Since the priority queue is sorted by score upper bound, the node representing o_1 contains the value 2.6, in addition to a pointer back to object o_1.

A bidirectional arrow connects o_1 and the corresponding node in the priority queue (Figure 3). In fact, each of the pointers included in an object (Section 4.1) references the node in a priority queue that represents such object. These pointers let us identify in constant time whether a given object is included in some priority queue, and if so, which node represents the object. Therefore, we can effectively delete and modify arbitrary objects in logarithmic time, without an expensive search for the object id in all priority queues.

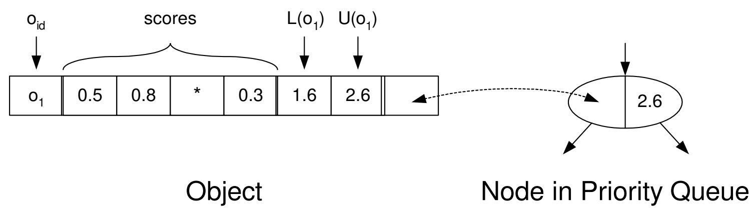


Figure 3: Representation of an object and the corresponding node in a priority queue.

Whenever possible, we use the simple priority queues described above to implement the Section 3 algorithms (e.g., to maintain the list of candidate objects sorted by U value in algorithm *Upper*). Unfortunately, this simple data structure is not sufficient to implement all our algorithms efficiently. We describe additional data structures that we require next.

Bounded Priority Queues: Consider the variations of the TA algorithm of Section 3.3. These algorithms maintain a list of the k objects with the largest lower bounds seen at any given time in the execution. To decide whether a new object should be included in the list (hence replacing an existing object), we cannot just compare the new object against the top element in the list, i.e., the one with the largest score lower bound. Instead, we need to compare the new object against the object in the list with the smallest lower bound, i.e., the object with the k-th lower bound seen so far (see Step (1-d) in algorithm *TA-EP*). To support this operation efficiently, we can use a simple adaptation of the priority queues described above that still provides constant-time access to the object with the k-th lower bound and logarithmic time for insertions and deletions. Moreover, this adapted priority queue can be implemented with a bounded amount of memory (proportional to k) that is independent of the size of the data sets.

We use a bounded priority queue (with capacity k) sorted in the *inverse order* of the intended one. Therefore, the smallest of the k elements in the priority queue can be accessed in constant time. We modified the insertion algorithm in the following way. Suppose we want to insert object o in the priority queue. We first compare o with the object o_t at the top of the queue, which has the k-th lowest lower bound seen so far. If o's lower bound is lower than that of o_t, we know that o is not among the top-k elements, so we discard it. Otherwise, we remove o_t from the priority queue and replace it by inserting o, thus maintaining the invariant that the priority queue contains the top-k seen objects according to their lower bound.

Combined Priority Queues: In some cases, the bounded priority queues described above are not appropriate to maintain the top-k objects seen at any given time in the execution. Specifically, sometimes the value that is used to rank the objects can increase or decrease during the execution of the algorithms. Consider the following objects and their associated expected scores: $(o_1, 10)$, $(o_2, 15)$, $(o_3, 12)$, $(o_4, 5)$, $(o_5, 9)$ and suppose that we want to keep track of the 3^{rd} highest expected score. (Such functionality is needed for example in the different *SelectBestSource* functions to get the value $score'_k$.) A bounded priority queue with capacity three as described before will contain objects o_1, o_2, and o_3, but not objects o_4 and o_5. Now suppose that we learn, after a random probe, that the new expected score of object o_1 is 8. In this case, it is not enough to restructure the priority queue by removing object o_1, changing its expected score and inserting it again. In fact, object o_5, with an expected score of 9, should replace object o_1 in the priority queue. Unfortunately, we discarded object o_5 in a previous step. This example shows that when the information used to sort the list of top objects can decrease, we need to keep track of all known objects.

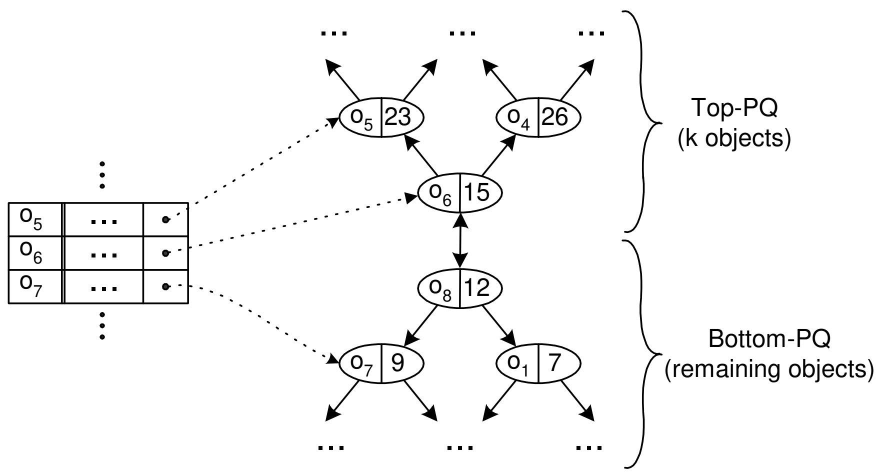


Figure 4: A combined priority queue keeps track of the object with the *k-th* largest expected score.

For this purpose, we implement a data structure that combines the standard and bounded priority queues. The new data structure, which we call *combined priority queue*, consists of two synchronized priority queues. The first one, denoted *Top-PQ*, is a bounded priority queue that keeps track of the current top-k objects. The second, denoted *Bottom-PQ*, is a standard priority queue that keeps track of the remaining objects seen at some point. The invariants in the combined priority queue are: all the objects in *Top-PQ* have a larger expected score than any object in *Bottom-PQ*, and the size of *Top-PQ* is equal to k whenever there are at least k objects in the combined queue. Figure 4 shows an instance of a combined priority queue. We implement the following procedures on top of the functions supported by the priority queues:

Top: The top of the combined priority queue is defined as the top of *Top-PQ*, i.e., the *k-th* element of the list. For instance, o_6 is the *k-th* element in Figure 4.

Insert object o: If the score of o is smaller than that of the *k-th* element of the list (the top score of *Top-PQ*), we insert o into *Bottom-PQ*. Otherwise (o is one of the top-k elements in the list), we (1) remove the top element from *Top-PQ* and insert it into *Bottom-PQ*, and (2) insert o into *Top-PQ*. For instance, to insert element o_{10} with score 18 to the combined queue in the figure, we first "move" object o_6 from *Top-PQ* to *Bottom-PQ* and then insert o_{10} into *Top-PQ*. As a result, the data structure invariant is maintained.

Delete object o: If o belongs to *Bottom-PQ*, we simply delete it. Otherwise (o is one of the top-k objects in the list), we (1) remove o from *Top-PQ*, and (2) remove the top element from *Bottom-PQ* and insert it in *Top-PQ*. For instance, after removing object o_4 from the combined queue, we need to "move" object o_8 from *Bottom-PQ* to *Top-PQ*. As a result, the data structure invariant is maintained.

It is easy to verify that both the insertion and deletion of objects take time logarithmic in the number of elements in the combined priority queue, and the identification of the *k-th* object takes constant time.

5 Cost Analysis

We now discuss the efficiency of our algorithms both in terms of time and space. For our analysis, we divide the algorithms into two families: the *Upper* family, which includes all variations of the *Upper* algorithm, and the *TA-Adapt* family, which consists of all variations of the *TA-Adapt* algorithm. Sections 5.1 through 5.3 address the execution time of the two families of algorithms: Section 5.1 shows that both *Upper* and *TA-Adapt* perform the smallest number of sorted accesses that any correct top-k query processing algorithm can indeed do. Then, Section 5.2 analyzes the number of random accesses that each algorithm requires. To complete the execution time analysis, Section 5.3 studies the local processing time of the algorithms. Finally, Section 5.4 discusses the space requirements of each algorithm.

5.1 Number of Sorted Accesses

The number of sorted accesses that an algorithm makes determines the number of objects that the algorithm considers as candidates to be in the top-k query result. Any correct algorithm that does not rely on "wild guesses" must perform sufficiently many sorted accesses to return the top-k solution with certainty. We now show that the algorithms in the *Upper* and *TA-Adapt* families all perform the smallest number of sorted accesses that any correct algorithm can do when only one *S-Source* is available:

Theorem 1: *Consider a top-k query q over one S-Source S and multiple R-Sources. Then, all variations of algorithms* Upper *and* TA-Adapt *from Section 3 retrieve exactly the smallest number of objects from S that any correct algorithm needs in order to answer q without "wild guesses."*

Proof: Assume that there is a hypothetical top-k query processing strategy W (for "wrong") that retrieves fewer objects than *TA-Adapt* does, without "wild guesses". Consider a query q over attributes $A_0, A_1, \ldots, A_n$, where A_0 is associated with *S-Source* S. Let $t_1, \ldots, t_j$ be the objects that one particular execution of W retrieves from S in sorted order. Furthermore, suppose that *TA-Adapt* retrieves more objects for q from S than W does. Specifically, the stopping condition of step 1 of *TA-Adapt* is not satisfied after retrieving object t_j from S. Then, at this point the threshold $T = U(t_j)$ is larger than the actual score of some of the top-k objects in $t_1, \ldots, t_{j-1}$. (*TA-Adapt* completely probes the final score of an object as soon as it is retrieved via sorted access.) Let $t_{i_1}, \ldots, t_{i_k}$ be the top-k objects returned by W as the result for q. Since W does not rely on "wild guesses," it follows that $t_{i_m} \in \{t_1, \ldots, t_j\}$, for $m = 1, \ldots, k$. Then, it is the case that $T > Score(q, t_{i_m})$ for at least one $m \in \{1, \ldots, k\}$. Now, let t_{j+1} be the top object from S not retrieved by W. (t_{j+1} is retrieved by *TA-Adapt* because the algorithm's stopping condition does not hold after t_j is accessed.) Assume that $Score_{A_0}(q, t_{j+1}) = Score_{A_0}(q, t_j)$. (This scenario is consistent with all scores "observed" by W and *TA-Adapt*.) Then, before probing t_{j+1} on any of the *R-Sources*, $U(t_{j+1}) = T$, and we know that $T > Score(q, t_{i_m})$ for at least one object t_{i_m} returned by W as part of the top-k answer for q. Then, if $Score_{A_i}(q, t_{j+1}) = 1\ \forall i = 1, \ldots, n$, the final score of object t_{j+1} for the query is better than that of object t_{i_m} (i.e., $Score(q, t_{j+1}) = T > Score(q, t_{i_m})$). (Again, this scenario is consistent with all scores "observed" by W and *TA-Adapt*.) Then, in this case W returned an incorrect answer for the query. In summary, any algorithm that does not rely on wild guesses and that retrieves fewer objects from the only *S-Source* than *TA-Adapt* does is incorrect.

We now show that the conditions to access the *S-Source* S for the variations of *TA-Adapt* and *Upper* are equivalent. *TA-Adapt* keeps accessing S as long as the score upper bound of the unseen objects, U_{unseen}, is greater than the score of any of the current top-k objects. Using *TA-Adapt*, the top-k scores among all previously retrieved objects are known since *TA-Adapt* completely probes all objects it discovers. (The variations of *TA-Adapt* presented in Section 3.3 might discard objects faster, but do not change the sorted-access condition.) *Upper* keeps accessing S as long as the score upper bound of some of the top objects retrieved is lower than U_{unseen}, which means that the score of some of the top retrieved objects is lower than U_{unseen}, and is hence equivalent to the *TA-Adapt* condition for accessing S. Therefore, *TA-Adapt* and *Upper* perform the same number of sorted accesses. (The variations of *Upper* presented in Section 3.2.2 differ only in the random accesses that they perform.)∎

5.2 Number of Random Accesses

In the previous section, we showed that the *TA-Adapt* and *Upper* algorithms perform the same number of sorted accesses. To decide which algorithm is the most efficient, we have to also consider the number of random accesses required by each algorithm.

As presented in the expanded version of [10], TA_z is "instance optimal", where "instance optimality" is defined as follows: [11]

Definition 3: [**Instance Optimality**] *Let $\mathcal{A}$ be a class of algorithms and $\mathcal{D}$ be a class of source instances. An algorithm $B \in \mathcal{A}$ is* instance optimal *over $\mathcal{A}$ and $\mathcal{D}$ if there are constants c and c' such that for every $A \in \mathcal{A}$ and $D \in \mathcal{D}$ we have that $cost(B, D) \leq c \cdot cost(A, D) + c'$, where $cost(A, D)$ is, in our context, the combined sorted- and random-access time insumed by algorithm A over the sources in D.*

[11] We slightly adapted the definition in [10] to use the terminology of our paper.

As discussed in Section 3.3, our *TA-Adapt* algorithm is a special case of TA_z and thus is "instance optimal" in the same sense.

An interesting observation is that the number of random accesses in *TA-Adapt* is an upper bound on the number of random accesses in *TA-Opt* and *TA-EP*: these two algorithms are optimizations over *TA-Adapt* aimed at reducing the number of random accesses. The shortcuts used in *TA-Opt* and *TA-EP* are only used to discard objects sooner than in *TA-Adapt* and do not affect the number of sorted accesses performed by the algorithms. Then, in the worst case, *TA-Opt* and *TA-EP* are equal to *TA-Adapt*. More interestingly, from Section 5.1, we know that the *Upper* algorithms perform the same number of sorted accesses as *TA-Adapt*, and therefore consider the same number of objects. Since *TA-Adapt* performs *all* random accesses for the objects considered, the *Upper* algorithms, in the worst case, do as many random accesses as *TA-Adapt*. Hence, the *TA-Adapt* "instance optimality" also applies to the *TA-Opt*, *TA-EP*, and *Upper* algorithms. Therefore, the experimental section of the paper (Section 7), in which we compare the *TA-Adapt* and *Upper* algorithms, will evaluate the algorithms with real-world and synthetic data to measure their "absolute" efficiency (they are all "instance optimal").

5.3 Running Time

In the time-complexity analysis we identify two components: the local running time used at each step in the algorithms, and the external time incurred in probing the different sources. As we will see, *TA-Adapt* is *locally* more efficient, in the sense that it can process each object-source pair with less overhead than *Upper*. However, as we show experimentally in Section 7, *Upper* is *globally* more efficient. That is, if we consider a complete execution of both algorithms, *Upper* results in considerable smaller execution times than *TA-Adapt* because it probes fewer random sources, which is likely to be the main bottleneck of any technique dealing with remote sources. In fact, the design of *Upper* sacrifices local efficiency to minimize the number of expensive random accesses needed to answer a top-k query. We now analyze the processing time of both families of algorithms.

Our implementation of *Upper* uses two main priority queues: a standard priority queue that maintains the set of candidate objects sorted by their score upper bound, and a combined priority queue that keeps track of the candidate object with the *k-th* expected score. At each iteration of the algorithm, we either retrieve a new object from the sorted source, or probe a random source for the object with the largest upper bound. In both cases we need to maintain the priority queues, either by adding a new object, or by modifying the relative position of an existing object in the sorted list. As shown in Section 4, we can perform these operations in time that is logarithmic in the number of already processed candidate objects. If n_S is the number of sorted accesses and n_R is the number of random probes, the local processing time to maintain the priority queues for *Upper* is $\mathcal{O}((n_S + n_R) \cdot \log(n_S))$, since n_S is an upper bound on the number of candidate objects at each iteration of the algorithm. The local processing time of *Upper* also includes the time spent in the *SelectBestSource* function

(Section 3.2.2), which depends on the method selected. For instance, the function in *Upper-Greedy* is $\mathcal{O}(n)$, and $\mathcal{O}(2^n)$ in *Upper-Filter* and *Upper-Subset*, where n is the number of *R-Sources*. The local processing time of *Upper-Greedy* is then $\mathcal{O}((n_S + n_R) \cdot \log(n_S) + n \cdot n_R)$ and the local processing time of *Upper-Filter* and *Upper-Subset* is $\mathcal{O}((n_S + n_R) \cdot \log(n_S) + 2^n \cdot n_R)$.

For *TA-Adapt*, we just maintain a bounded priority queue q that keeps track of the top k objects retrieved. At each step of the algorithm, we retrieve a new element from the sorted source (step 1a). After probing all needed random sources, we update q to include the current element (if necessary). If n_S is the number of sorted accesses and n_R is the number of random probes, the local processing time for *TA-Adapt* is $\mathcal{O}(n_S \cdot \log(k) + n_R)$.

As discussed in Section 5.1, the number of sorted accesses n_S is the same for both algorithms. Also, $k \leq n_S$ (we need to retrieve at least k objects to return the top-k). If we assume that $n_R^{Upper} = n_R^{TA-Adapt}$, then we have the following inequality: ${t_{local}}_{TA-Adapt} \leq {t_{local}}_{Upper}$, where t_{local} is the local processing time. In other words, *TA-Adapt* is asymptotically faster than *Upper* if we just consider the local processing time. However, to get the total time taken by any algorithm, we need to add to the local processing time the time $tS(S) \cdot n_S + \sum tR(R_i) \cdot n_i$, where n_i is the number of random accesses to *R-Source* R_i, $tR(R_i)$ is the time taken to do a random access to R_i, and $tS(S)$ is the time taken to do a sorted access to *S-Source* S. Since the values of $tS(S)$ and $tR(R_i)$ are large in comparison to the local processing time, and, as we will show experimentally, $n_R^{Upper} < n_R^{TA-Adapt}$, for real-world scenarios *Upper* results in considerable faster executions than *TA-Adapt*.

5.4 Space Requirements

We now analyze the space required by both families of algorithms. As explained in the previous section, *Upper* uses two priority queues that grow linearly with the number of sorted accesses (both priority queues represent all candidate objects seen at a given point in time). Therefore, the space needed for *Upper* is $\mathcal{O}(n_S)$. In contrast, since *TA-Adapt* only uses a bounded priority queue, the space needed is just $\mathcal{O}(k)$, independent of the number of sorted accesses (i.e., *TA-Adapt* can be implemented using bounded buffers). Although *TA-Adapt* requires less memory than *Upper*, *TA-Adapt* cannot be used as an incremental algorithm. In fact, if after retrieving the top-k objects using *TA-Adapt* we decide we want the *next* k objects, we have to start *TA-Adapt* from scratch with parameter $2k$. In contrast, we can easily modify the queues in *Upper* on the fly to get the *next* k objects.

6 Evaluation Setting

In the previous section, we showed that the number of sorted accesses is the same for all versions of *TA-Adapt* and *Upper*. We also showed that the local running time is higher for *Upper* than for *TA-Adapt*. However, we have not determined the relative performance of the *Upper* and *TA-Adapt* algorithms in terms of random accesses. Since random accesses are likely to be expensive, we now turn to evaluating the time spent in random accesses experimentally to compare the total processing time of the different techniques. In this section, we describe the synthetic data sets (Section 6.1) we use to evaluate the strategies of Section 3, as well as the prototype we implemented to test our strategies over real web-accessible sources (Section 6.2). Finally, we discuss the metrics and other settings that we use in our experimental evaluation (Section 6.3).

6.1 Synthetic Sources

We generate different synthetic data sets. Objects in these data sets have attributes from a single *S-Source* S and several *R-Sources* (the default number of *R-Sources* is five). The data sets vary in the number of objects in *Objects*(S) and in the correlation between attributes and their distribution. Specifically, given a query, we generate individual attribute scores for each conceptual object in our synthetic database in three ways:

- *Uniform* data set: We assume that attributes are independent of each other and that scores are uniformly distributed (default setting).

- *Correlation* data set: We assume that attributes exhibit different degrees of correlation, modeled by a correlation factor *cf* that ranges between -1 and 1 and that defines the correlation between the *S-Source* and the *R-Source* scores. Specifically, when *cf* is zero attributes are independent of each other. Higher values of *cf* result in positive correlation between the *S-Source* and the *R-Source* scores, with all scores being equal in the extreme case when *cf*=1. In contrast, when *cf*<0, the *S-Source* scores are negatively correlated with the *R-Source* scores.

- *Gaussian* data set: We generate the multiattribute score distribution by producing five overlapping multidimensional Gaussian bells [20].

The random-access time for each *R-Source* R_i (i.e., $tR(R_i)$) is a randomly generated integer ranging between 1 and 10, while the sorted-access time for *S-Source* S (i.e., $tS(S)$) is randomly picked from $\{0.1, 0.2, \ldots, 1.0\}$.

6.2 Real Web-Accessible Sources

We implemented a prototype in Python [12] to evaluate our strategies over real web-accessible sources. The prototype implements (an expanded version of) our restaurant example of Section 2. Users input a starting address, the type of cuisine in which they are interested (if any), and importance weights for the following *R-Source* attributes: *SubwayTime* (handled by the SubwayNavigator site [13]), *DrivingTime* (handled by the MapQuest site), *Popularity* (handled by the AltaVista search engine [14]; see below), *ZFood*, *ZService*, *ZDecor*, and *ZPrice* (handled by the Zagat Review web site), and *TRating* and *TPrice* (provided by the New York

[12] `http://www.python.org`
[13] `http://www.subwaynavigator.com`
[14] `http://www.altavista.com`

Source	*Attribute(s)*	*Input*
Verizon Yellow Pages (S)	*Distance*	type of cuisine, user address
Subway Navigator (R)	*SubwayTime*	restaurant address, user address
MapQuest (R)	*DrivingTime*	restaurant address, user address
AltaVista (R)	*Popularity*	free text with restaurant name and address
Zagat Review (R)	*ZFood*, *ZService* *ZDecor*, *ZPrice*	restaurant name
NYT Review (R)	*TRating*, *TPrice*	restaurant name

Table 1: Real web-accessible sources used in the experimental evaluation.

Times at the New York Today web site). The Verizon Yellow Pages listing [15], which returns restaurants of the user-specified type sorted by shortest distance from a given address, is the only *S-Source*. Table 1 summarizes these sources and their interfaces.

Popularity Attribute: The *Popularity* attribute requires further explanation. We approximate the "popularity" of a restaurant with the number of web pages that mention the restaurant, as reported by the AltaVista search engine. (The idea of using web search engines as a "popularity oracle" has been used before in the WSQ/DSQ system [11].) Consider, for example, restaurant "Tavern on the Green," which is one of the most popular restaurants in the United States. A query on AltaVista on "Tavern on the Green" AND "New York" returns 2,326 hits. In contrast, the corresponding query for a much less popular restaurant on New York City's Upper West Side, "Caffe Taci" AND "New York," returns only five hits. Of course, the reported number of hits might inaccurately capture the actual number of pages that talk about the restaurants in question, due to both false positives and false negatives. Also, in rare cases web presence might not reflect actual "popularity." However, anecdotal observations indicate that search engines work well as coarse popularity oracles.

Naturally, the real sources above do not fit our model of Section 2 perfectly. For example, some of these sources return scores for multiple attributes simultaneously (e.g., the Zagat Review site). Also, as we mentioned before, information on a restaurant might be missing in some sources (e.g., a restaurant might not have an entry at the Zagat Review site). In such a case, our system will give a default (expected) score of 0.5 to the score of the corresponding attribute.

Adaptive Time: In a real web environment, source access times are usually not fixed and depend on several parameters such as network traffic or server load. Using a fixed approximation of the source response time (such as an average of past response times) may result in degraded performance since our algorithms use these times to choose what probe to do next.

[15] `http://www.superpages.com`

Parameter	k	$\|S\|$	n	*Data Set*
Default Value	50	10,000	5	*Uniform*

Table 2: Default setting of some experiment parameters for synthetic sources.

To develop accurate adaptive estimates for the tR times, we adapt techniques for estimating the round trip time of network packets. Specifically, TCP implementations use a *"smoothed" round trip time estimate* ($SRTT$) to predict future round trip times, computed as follows:

$$SRTT_{i+1} = (\alpha \times SRTT_i) + ((1 - \alpha) \times s_i)$$

where $SRTT_{i+1}$ is the new estimate of the round trip time, $SRTT_i$ is the current estimate of the round trip time, s_i is the time taken by the last round trip sample, and α is a constant between 0 and 1 that controls the sensitivity of the $SRTT$ to changes. For better performance, Mills [16] recommends using two values for α: $\alpha = 15/16$, when the last sample time is lower than the estimate time ($SRTT_i$), and $\alpha = 3/4$, when the last sample time is higher than the estimate. This makes the estimate more responsive to increases in the source response time than to decreases. Our prototype keeps track of the response time of probes to each *R-Source* R_i and adjusts the average access time for R_i, $tR(R_i)$, using the $SRTT$ estimates above. Since the accesses to the *S-Source* S are decided independently of its access time, we do not adjust $tS(S)$.

6.3 Other Experimental Settings

Our query processing strategies attempt to minimize the total processing time for top-k queries, both for random and for sorted access to the sources. To measure the relative performance of the techniques over an *S-Source* S and *R-Sources* $R_1, \ldots, R_n$, we use the following metric:

$$t_{probes} = n_S \cdot tS(S) + \sum_{i=1}^{n} n_i \cdot tR(R_i)$$

where n_S is the number of objects extracted from *S-Source* S, n_i is the number of random-access probes for *R-Source* R_i, and tS and tR are as specified in Definition 2. t_{probes} then approximates the execution time for a query without counting local processing time (Section 5.3). We also report results on the local processing time t_{local} of the techniques, and on the overall time $t_{total} = t_{local} + t_{probes}$, which includes both local execution and probing times.

For the synthetic data sets and for each setting of the experiment parameters, we generate 100 queries randomly with their associated weights, and compute the average t_{local} and t_{probes} values. We report results for top-k queries for different values of k, $|S|$, cf and for various assignments of weights and times to sources. In the default setting, $k = 50$ (i.e., queries ask for the best 50 objects), $|S| = 10,000$, and we use the *Uniform* data set. The default setting values are summarized in Table 2.

For the real data sets, we use four queries, specifying addresses in three different Manhattan neighborhoods and different restaurant type preferences. Attributes *Distance*, *SubwayTime*, *DrivingTime*, *ZFood*, *ZService*, *ZDecor*, and *TRating* have "default" target values in the queries (e.g., a *DrivingTime* of 0 and a *ZFood* rating of 30). The target value for *Popularity* is arbitrarily set to 100 hits, while *ZPrice* and *TPrice* are set to the least expensive value in the scale. In all four queries, the weight of the *S-Source* attribute (i.e., *Distance*) is roughly twice the weight of any *R-Source* attribute. *R-Sources* access times are reevaluated during query processing using the adaptive estimates presented in Section 6.2.

Next, we experimentally compare the algorithms that we discussed in Section 3, namely *TA-Adapt* and *TA-Opt* (Section 3.3.1), *TA-EP* (Section 3.3.2), and *Upper* (Section 3.2.2). For all experiments we ran the three versions of *Upper* introduced in Section 3, i.e., *Upper-Greedy*, *Upper-Filter*, and *Upper-Subset*. In general, the three versions of *Upper* performed similarly. However, *Upper-Filter* resulted in the best performance among all versions for virtually all cases. Therefore, we report results only for that version of *Upper*. Hence, in the next section, we refer to *Upper-Filter* simply as *Upper*. We also report results for the *Optimal* technique of Section 3.2.1. As discussed, this technique is only of theoretical interest, and serves as a lower bound for the time that any strategy without "wild guesses" would take to process top-k queries.

7 Evaluation Results

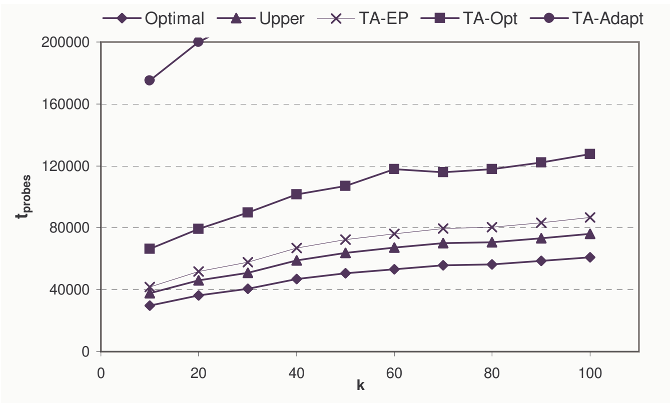
We now present the experimental results for the techniques of Section 3, using the data sets and general settings described in Section 6.

7.1 Results for Synthetic Data Sets

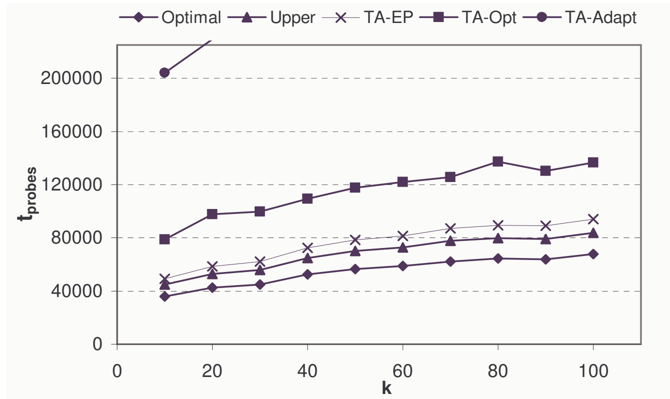
We first study the performance of the techniques when we vary the synthetic data set parameters.

Effect of the Number of Objects Requested k: Figure 5 reports results for the default setting (Table 2), as a function of k and for both the *Uniform* and *Gaussian* synthetic data sets. As k increases, the time needed by each algorithm to return the top-k objects increases as well, since all techniques need to retrieve and process more objects. The *Upper* strategy consistently outperforms all other techniques, with average t_{probes} time close to that of the theoretical lower bound, *Optimal*. We can see that our optimizations over *TA-Adapt*, namely *TA-Opt* and *TA-EP*, result in dramatic improvements in performance over *TA-Adapt* due to savings in random accesses. We then remove *TA-Adapt* from further consideration in the remaining discussion.

Effect of the Number of *R-Source* Sources: Figure 6 reports results for the default setting, as a function of the total number of *R-Sources* for both *Uniform* data sets. Not surprisingly, the t_{probes} time needed by all the algorithms increases with the number of available sources. When we consider a single *S-Source* and a



(a) *Uniform data set*



(b) *Gaussian data set*

Figure 5: Performance of the different strategies for the default setting of the experiment parameters, as a function of the number of objects requested k, and for two synthetic data-set distributions.

single *R-Source*, t_{probes} is almost the same for all algorithms. However, when more *R-Sources* are available, the differences between the techniques become more pronounced, with *Upper* consistently resulting in the best performance.

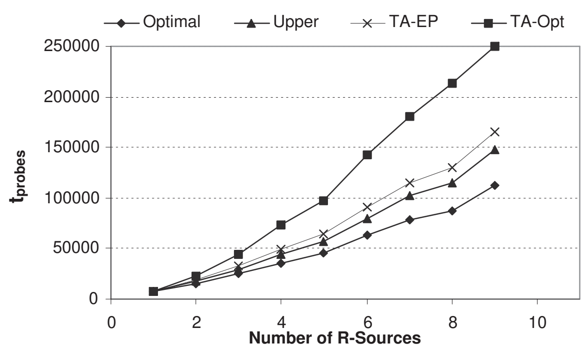


Figure 6: Performance of the different strategies for the *Uniform* data set, as a function of the number of *R-Sources*.

Effect of the Number of Objects in *S-Source S*: Figure 7 studies the impact of the size of *S-Source S*. As the number of objects increases, the performance of each algorithm drops since more objects have to be evaluated before a solution is returned. The t_{probes} time needed by each algorithm is approximately linear in the number of objects in S. *Upper* gives faster execution results and scales better than the other techniques since it only considers objects that need to be probed before the top-k answer is reached and therefore does not waste resources on useless probes.

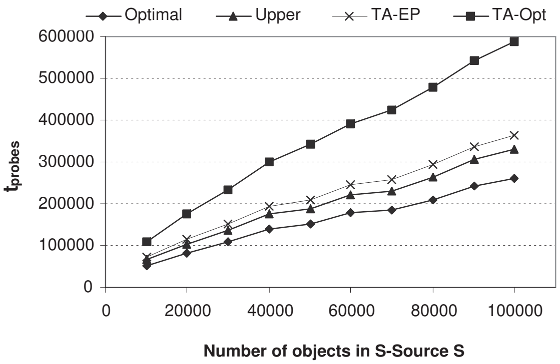
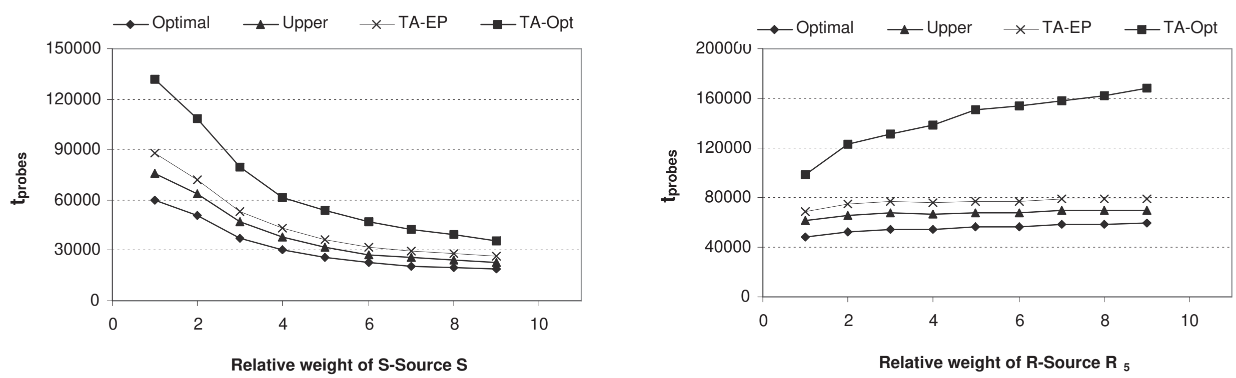


Figure 7: Performance of the different strategies for the *Uniform* data set, as a function of the number of objects in *S-Source S*.

Effect of Attribute Weights: Figure 8 reports results on the impact of attribute weights on the t_{probes} time. We vary the weight of the *S-Source S* (Figure 8(a)) and the *R-Source* R_5 (Figure 8(b)) relative to the weight of the remaining sources. In particular, we set the varying weight as a multiple of the average of the remaining weights. Figure 8(a) shows that all techniques improve their t_{probes} times when the weight of the *S-Source* attribute is high, since fewer random probes are needed to identify the top-k objects. Also, the performance of *TA-Opt* degrades as the weight of *R-Source* R_5 increases (Figure 8(b)): *TA-Opt* does not use any information about the relative weights of the sources to order the random probes, in contrast to the other techniques. We note that we obtained analogous results when we varied the access times of the different sources instead of their weights.



(a) Varying the relative weight of the *S-Source* (b) Varying the relative weight of an *R-Source*

Figure 8: Performance of the different strategies for the *Uniform* data set, and for various attribute-weight combinations.

Effect of Attribute Correlation: We now turn to the *Correlation* data set (Section 6) and evaluate the effect of attribute correlation on the performance of the query processing techniques. Figure 9 shows that when the correlation factor *cf* is high and positive the performance of all techniques improves dramatically. Interestingly, a negative correlation between the *R-Sources* and the *S-Source* attribute scores significantly affects the performance of the *TA-Adapt* algorithms. For correlation factors close to -1, the order of the objects in the *S-Source* is close to the inverse of the order by final scores. Therefore, both *TA-Opt* and *TA-EP* need to probe each object almost completely before proceeding to the next one, and have to consider almost all objects in *S-Source S* before producing the final answer, which results in significantly larger t_{probes} times compared to *Upper*.

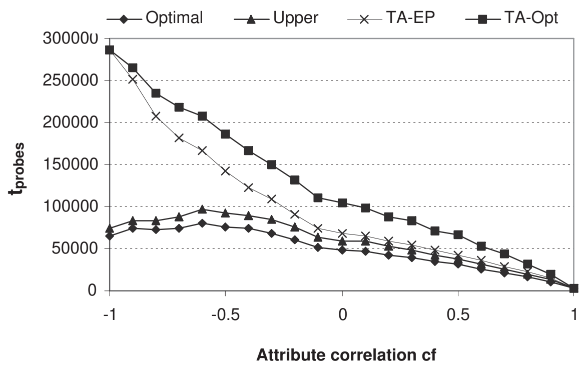


Figure 9: Performance of the different strategies for the *Correlation* data set, as a function of the correlation factor *cf*.

To further understand the effect of attribute correlation on performance, we also generated data sets in which groups of sources were correlated. In Figure 10(a) we show the performance of the different algorithms when a varying fraction of the six available *R-Sources* was correlated to the *S-Source*, with the remaining *R-Sources* being inversely correlated to the *S-Source*. When all six *R-Sources* are correlated (or inversely correlated) to the *S-Source*, the situation is similar to that of Figure 9 for cf near zero and one. The hardest case for the *Optimal* algorithm is when half of the *R-Source*s are correlated and the other half is inversely correlated to the *S-Source*. Even in this case, *Upper* performs significantly better than the *TA-Adapt* variations and is close to the optimal performance. Finally, we divided the *R-Sources* in two groups so that the scores of objects from *R-Sources* in the same group was correlated (and uncorrelated to the *S-Source*). Figure 10(b) shows the performance of the techniques when the groups have (1,5), (2,4), and (3,3) *R-Sources* each. The results are consistent with previous experiments. When the number of correlated sources is high, it is easier to discard objects, and the algorithms have better performance.

Effect of Varying Expected Scores: In absence of reliable information on source-score distribution, our techniques approximate expected scores with the constant 0.5 (see Section 3). This estimation can result in bad

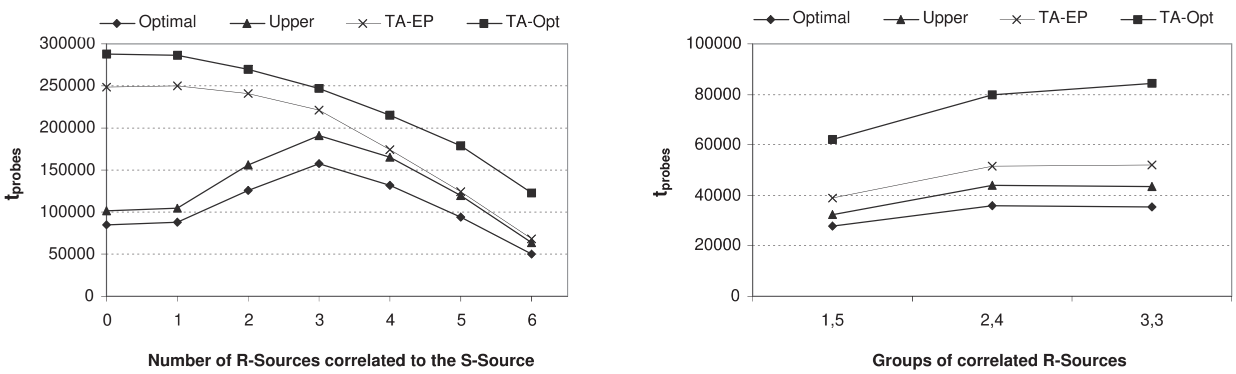


Figure 10: Performance of the different strategies for various degrees of correlation among sources.

performance when the actual expected scores are far from 0.5. To evaluate the effect of this choice of expected scores on the performance of *Upper*, we generated data sets with different score distributions and compared the performance of *Upper* with and without knowledge of the actual expected scores. In particular, we first evaluated 100 queries using *Optimal*, *Upper*, and *TA-EP* assuming that the expected scores were 0.5. Then, we evaluated the same queries, but this time we let *Upper* use the actual expected scores to choose which sources to probe. We refer to this version of *Upper* as *Upper-E*. (Note that *Optimal* and *TA-EP* do not rely on expected scores.) The results are shown in Figure 11. For the first experiment, three out of the five *R-Sources* had scores uniformly distributed between 0 and 1 (with expected score 0.5), the fourth *R-Source* had scores varying from 0 to 0.2 (with expected score 0.1), and the fifth *R-Source* had scores ranging from 0.8 to 1 (with expected score 0.9). For the second experiment, the expected scores for the *R-Sources* were random values between 0 and 1. Not surprisingly, *Upper-E* results in smaller t_{probes} time than *Upper* (and close to the *Optimal* one), showing that *Upper* can effectively take advantage of any extra information about expected sources in its *SelectBestSource* routine. In any case, it is important to note that the performance of *Upper* is still better than that of *TA-EP* even when *Upper* uses the default value of 0.5 as the expected attribute score.

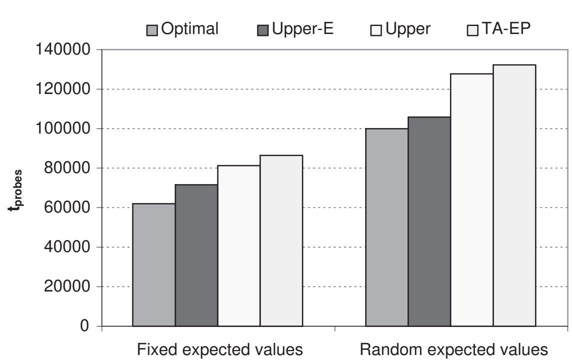


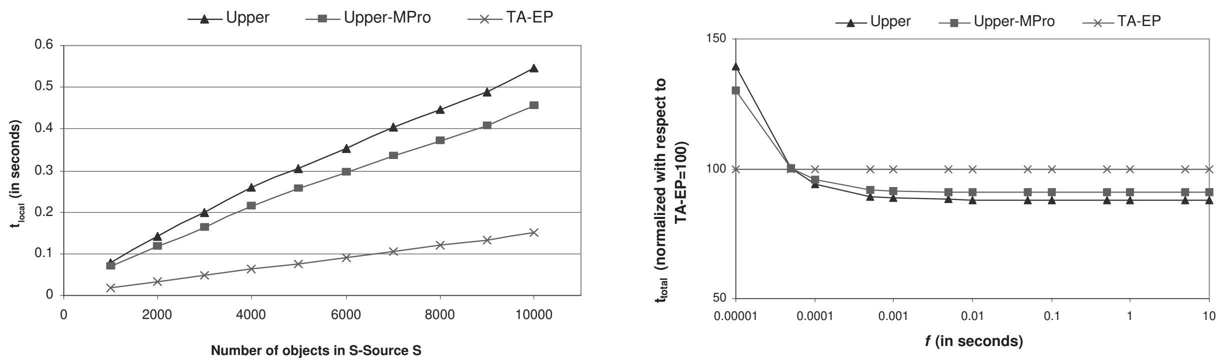
Figure 11: The performance of *Upper* improves when the expected scores are known in advance.

7.2 Local Processing Time

In Section 5.3 we showed that the local time t_{local} of the *TA-EP* algorithm is asymptotically smaller than that of *Upper*. However, as we saw in the previous sections, *Upper* results in much fewer *R-Source* probes, which are expensive. We now show experimentally that *Upper* results in considerable overall faster execution times than *TA-EP*.

In this section, we also consider a variation of *Upper* that uses a fixed schedule to select which source to access next in random access. This approach was introduced by Chang and Hwang [5], who presented the *MPro* algorithm for evaluating top-k queries over expensive predicates. *MPro* is similar to *Upper* in that it always probes the object with the highest score upper bound. However, a key difference between *MPro* and *Upper* is that *MPro* assumes a fixed schedule of sources to access in random access, and does not base its choices on the current query state. Additionally, *MPro* assumes that all accesses to the only sorted-access source are performed first, and does not interleave sorted and random accesses. In the remainder of the discussion, we will consider *Upper-MPro*, a variation of *Upper* that assumes a fixed schedule for random accesses, but takes advantage of possible interleaving of sorted and random accesses. *Upper-MPro* can be considered as an execution of *Upper* for which the *SelectBestSource* function returns sources in the same order for all objects, hence requiring less local processing. As in *TA-EP*, we order sources for random accesses according to their *Rank* value (see Section 3.3.2).

Figure 12(a) shows the t_{local} time for *Upper*, *Upper-MPro*, and *TA-EP* when using the default setting of the experiments in Table 2 and varying the number of elements in *S-Source* S. *TA-EP* is *locally* more efficient than both versions of *Upper* using around one third of the time of *Upper*, and *Upper-MPro* is slightly faster than *Upper* in terms of local processing time since the *SelectBestSource* computation is much simpler. However, the difference in local processing time between *TA-EP* and *Upper* is under half a second on average. If random accesses are fast, then the extra processing time required by *Upper* is likely not to pay off. In contrast, for real web sources, with high latencies, the extra local work is likely to result in faster overall executions. To understand this interaction between local processing time and random-access time, we vary the absolute value of the time "unit" f with which we measure the random-access time tR for the *R-Sources*. Figure 12(b) shows the total processing time of all three techniques for varying values of f (tR is randomly chosen between 1 and 10 time units), normalized with respect to the total processing time of *TA-EP*. From this figure, it follows that for *TA-EP* to be faster than *Upper* in total execution time, the time unit for random accesses should be less than 0.05 msecs, which translates in random access times no larger than 0.5 msecs. For comparison, note that the fastest real-web *R-Source* access time in our experiments was around 25 msecs. For all realistic values of f (i.e., 0.025 seconds or larger) it follows that while *TA-EP* is *locally* faster than *Upper*, *Upper* is *globally* more efficient. Additionally, Figure 12(b) shows that *Upper* outperforms *Upper-MPro* for f higher than 0.05 msecs, which means that the extra computation in the *SelectBestSource* function of *Upper* results in savings in probing time and thus in overall faster query execution times. Note that for high values of f, the local processing time of the techniques becomes negligible in comparison with the random-access time.



(a) The local processing time for *Upper*, *Upper-MPro*, and *TA-EP*, as a function of the number of objects in the *S-Source S*

(b) The total processing time for *Upper*, *Upper-MPro*, and *TA-EP*, as a function of the time scale f

Figure 12: Comparing local and total processing time for *Upper*, *Upper-MPro*, and *TA-EP*.

7.3 Results for Real Web-Accessible Data Sets

Our next results are for the six web-accessible sources, handling 10 attributes, which we described in Section 6.2 and summarized in Table 1. To model the initial access time for each source, we measured the response times for a number of queries at different hours and computed their average. We then issued four different queries to these sources and timed their total execution time. The source access time is adjusted at run time using the $SRTT$ value discussed in Section 6.2. Figure 13 shows the execution time for each of the queries, and for the *Upper*, *TA-EP*, and *TA-Opt* strategies. Just as for the synthetic data sets, our *Upper* strategy performs significantly better than the two variations of the *TA-Adapt* algorithm. Figure 13 shows that real-web queries have high execution time, which are a result of accessing the sources sequentially. Parallel versions of the algorithms discussed in this paper result in lower overall running times (see Section 10). (The *R-Sources* we used are slow with an average random access time of 1.5 seconds.)

In summary, our experimental results consistently show that *Upper* outperforms all other methods, with performance close to that of the *Optimal* technique. Furthermore, our modifications of the *TA-Adapt* algorithm, *TA-EP* in particular, resulted in significant improvements in performance. As a final observation, note that all the algorithms discussed in this paper correctly identify the top-k objects for a query according to a given scoring function. Hence there is no need to evaluate the "correctness" or "relevance" of the computed answers. However, the design of appropriate scoring functions for a specific application is an important problem that we do not address in this paper.

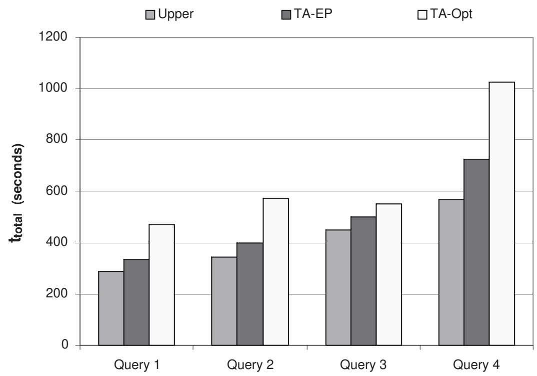


Figure 13: Experimental results for the real web-accessible data sets relevant to our New York City restaurant scenario.

S_1		S_2	
o_1	1	o_{100}	0.95
o_2	0.1	o_{99}	0.1
...	...	...	...
o_{99}	0.1	o_2	0.1
o_{100}	0.1	o_1	0

Table 3: A top-k query execution over two *S-Sources* ($k = 1$; $|Objects(S_1)| = |Objects(S_2)| = 100$; $w_{S1} = w_{S2} = 0.5$).

8 Relaxing the Source Model

In Section 3 we made a number of simplifying assumptions to present our query processing algorithms. Perhaps the most severe of these assumptions is the restriction that there be only one *S-Source* and arbitrarily many *R-Sources*. We now consider relaxations of this restriction: Section 8.1 discusses the implications of handling multiple *S-Sources* on query processing efficiency. Then, Section 8.2 presents an adaptation of our techniques to work over multiple *SR-Sources* and *R-Sources*. Finally, Section 8.3 reports the results of an experimental evaluation of this new algorithm.

8.1 Multiple *S-Sources*

In this paper we focus on algorithms that return the exact top-k objects for a query, together with their scores. So far, our discussion has been restricted to one *S-Source* and arbitrarily many *R-Sources*. A scenario with several *S-Sources* (with no random-access interface) is problematic: to return the top-k objects for a query together with their scores, as required by our query model, we might have to access *all* objects in some of the

S-Sources to retrieve the corresponding attribute score for a top-k object. This can be prohibitively expensive in practice.

Fagin et al. presented the *NRA* algorithm [10] to deal with multiple *S-Sources*. *NRA* only identifies the top-k objects and does not compute their final scores: if a top-k object t has a low score for one of the *S-Sources*, then *NRA* might output t without knowing t's score for this source. Identifying the top-k objects *without computing their final scores* can then be much cheaper than returning the top-k objects along with their scores. However, in some cases, even identifying the top-k objects over several *S-Sources* proves to be expensive. For example, consider the two *S-Sources* S_1 and S_2 in Table 3. A top-1 query over these two sources will have to access *all* objects in both sources to return o_{100} as the top object, since it is impossible to decide which of o_1 and o_{100} is best until both sources have been completely accessed. This will take time $|Objects(S_1)| \cdot tS(S_1) + |Objects(S_2)| \cdot tS(S_2)$.

Fagin et al.'s *NRA* algorithm could be easily adapted to a scenario with multiple *S-Sources*, *SR-Sources*, and *R-Sources*: upon discovery of an object under sorted access in one of the *S-Sources* or *SR-Sources*, all possible random accesses for the newly discovered object are performed. The stop condition is identical to *NRA*'s, and the algorithm returns the top-k objects *without their final scores* as soon as they are identified. The optimizations used in *TA-Opt* and *TA-EP* can be applied to this adaptation of *NRA*, to avoid performing useless random probes and improve query execution efficiency.

However, adapting *Upper* to a scenario with multiple *S-Sources* is not as simple: *Upper* focuses on the object t_H with the highest score upper bound. If t_H is not completely probed, *Upper* selects a random-access probe on t_H to perform next; in the presence of *S-Sources*, there might be no random-access probes left on t_H, even if t_H's score for the query is not completely defined. An adaptation of *Upper* would then require a potentially expensive strategy to retrieve needed information from the *S-Sources*. As a result, the adaptation of *NRA* discussed above appears as a more promising strategy for handling a multiple-*S-Source* scenario.

8.2 Multiple *SR-Sources*

So far, we have presented efficient top-k query processing algorithms over a single *S-Source* (or *SR-Source*) and multiple *R-Sources*. We will now discuss algorithms that can answer queries when multiple *SR-Sources* and *R-Sources* are available. Specifically, we describe adaptations of a recently proposed algorithm for multiple *SR-Sources* and *R-Sources* [10], as well as introduce two new variants of our *Upper* algorithm for this relaxed source scenario.

***TA_z-EP* and *TA_z-EP-Unbounded*:** *TA* sequentially accesses all *SR-Sources* in sorted access. When it discovers a new object, it randomly accesses every other source to compute the final score of the object. *TA_z* [10] is an adaptation of *TA* to the scenario in which some of the sources do not provide sorted-access capabilities: the sorted access phase will only consider *SR-Sources*, while the random accesses will be done over all sources. (As discussed, *TA-Adapt* is identical to TA_z when only one *S-Source* is present.) Our optimizations over *TA-Adapt*, namely *TA-Opt* and *TA-EP*, can still be used over TA_z. For space efficiency, TA_z assumes bounded buffers, which may lead to redundant random accesses, since TA_z does not keep track of all the objects it has already seen. In our experiments, we will present two versions of TA_z: one with bounded buffers (TA_z*-EP*), and one with unbounded buffers (TA_z*-EP-Unbounded*). The unbounded-buffers version of TA_z gains in efficiency by avoiding redundant random accesses, but requires space that is linear in the number of objects retrieved n_S. (Note that the *Upper* algorithms also use unbounded buffers and need $\mathcal{O}(n_S)$ space.)

***Upper-Weight*:** *Upper* is designed for one *S-Source* and several *R-Sources*. A simple way to adapt *Upper* so that it can operate on more than one *SR-Source* is to regard all sources but one as *R-Sources*. Hence one of the *SR-Sources* is chosen to be the one contacted in sorted access, while the other *SR-Sources* are only probed using random access. Different approaches can be used to pick which *SR-Source* to access in sorted access. In the experiments below, we report the performance for *Upper-Weight*, which chooses the *SR-Source* with the highest associated query weight as the *S-Source* and otherwise proceeds as *Upper-Filter* (Section 3.2.2).

***TA-Upper*:** The unbounded-buffers version of the *TA* algorithm can be slightly modified to output the top objects incrementally. The result of an execution of *TA* over several *SR-Sources* can then be regarded as a single *S-Source* for which the `getNext` interface would return the next top object. We can then define a hybrid strategy that uses *Upper* to combine the *R-Source* attribute scores with the result of the execution of *TA* over the *SR-Sources*. We call this technique *TA-Upper*.

***Upper-Relaxed*:** Ideas from both *TA* and *Upper* can be combined to create an algorithm that deals with multiple *SR-Sources* and *R-Sources*. The resulting algorithm is similar to *Upper* except that in the sorted access step (when unseen objects might have larger scores than all "candidate" objects), *SR-Sources* are accessed alternatively. The order in which *SR-Sources* are accessed in sorted access can be decided in several ways. The simplest approach is to use a round-robin algorithm. Güntzer et al. [12] presented a variation of *TA* that uses distribution information to order sorted accesses. Alternatively, we could access *SR-Sources* by decreasing order of a $Rank$ value similar to the one defined in Section 3.2.2. We define the $Rank$ of a sorted-access source S as $Rank(S) = \frac{w_S.(1-e(S))}{tS(S)}$, where $e(S)$ is the expected score of an object not yet seen under sorted access. We choose this last approach in the *Upper-Relaxed* algorithm that we present next. This algorithm returns the top-k objects for a query q over n_{sr} *SR-Sources* and n_r *R-Sources*. We define U_{unseen} as the upper bound of any object not seen under sorted access.

Algorithm Upper-Relaxed (Input: top-k query q)

1. Initialize $U_{unseen} = 1$, $Candidates = \emptyset$, and $returned = 0$.

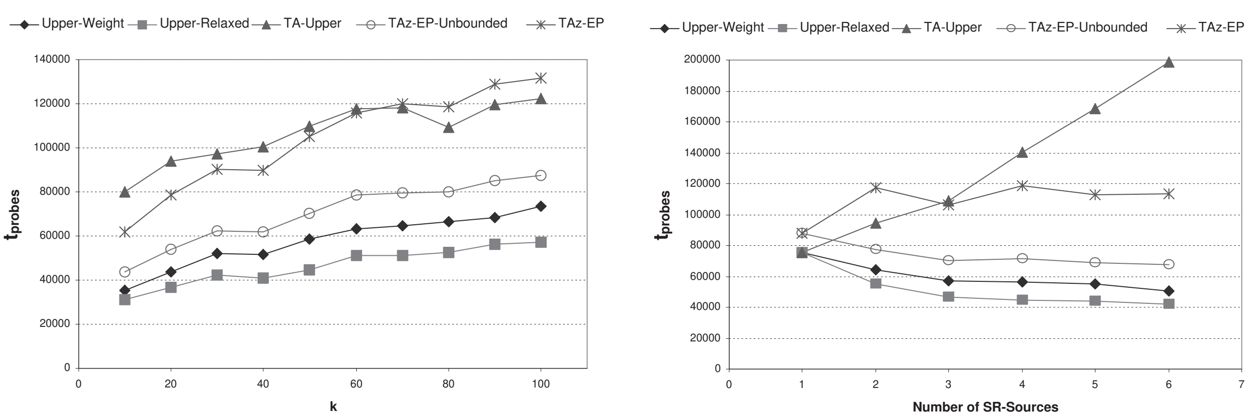
2. While $(returned < k)$
   (a) If $Candidates \neq \emptyset$, pick $t_H \in Candidates$ such that $U(t_H) = \max_{t \in Candidates} U(t)$.
   Else t_H is undefined.
   (b) If t_H is undefined or $U(t_H) < U_{unseen}$ (unseen objects might have larger scores than all candidates):
   - Choose an *SR-Source* S_i $(1 \leq i \leq n_{sr})$ to access next in sorted access such that $Rank(S_i) = max_{j=1}^{n_{sr}}\{Rank(S_j)\}$.
   - Get the best unretrieved object t from S_i: $(t, s_i) \leftarrow \texttt{getNext}_{S_i}(q_i)$.
   - Update $U_{unseen} = ScoreComb(s_\ell(1), \ldots, s_\ell(n_{sr}), \underbrace{1, \ldots, 1}_{n_r\ times})$, where $s_\ell(i)$ is the last value seen under sorted access in *SR-Source* S_i, and insert t into $Candidates$.

   Else If t_H is completely probed (t_H is one of the top-k objects):
   - Return t_H with its score; remove t_H from $Candidates$.
   - $returned = returned + 1$.

   Else:
   - $R_i \leftarrow$ *SelectBestSource*(t_H, *Candidates*).
   - Probe source R_i on object t_H: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t_H)$.

8.3 Experimental Results

We implemented the five algorithms presented in the previous section and evaluated them experimentally. For the evaluation, we used the synthetic data sets presented in Section 6.1. Our default setting consists of six sources, divided in three *SR-Sources* and three *R-Sources*. For each experiment, we ran 100 queries and averaged the resulting execution times.



(a) Varying the number of objects requested k (b) Varying the number of *SR-Sources*

Figure 14: Performance of the different strategies for the relaxed source setting.

We report the results for our five strategies in Figure 14. In Figure 14(a), we report the results as a function of the number of objects requested k. TA_z*-EP-Unbounded* performs better than TA_z*-EP* since it does not perform redundant random accesses. However, it requires more space than TA_z*-EP* because it needs unbounded buffers.

As expected, *Upper-Relaxed*, which is specifically designed for this relaxed source model, is more efficient than *Upper-Weight*. However, *Upper-Weight*, a simple adaptation of *Upper*, outperforms the remaining strategies. *TA-Upper* does not perform well, with probing times higher than those of TA_z*-EP*. In Figure 14(b), we vary the number of *SR-Sources*, while keeping the same total number of sources. Interestingly, both *Upper-Relaxed* and *Upper-Weight*, which interleave probes on objects, perform better than TA_z*-EP-Unbounded* and TA_z*-EP* even when all sources are *SR-Sources*, which is the original setting for the *TA* algorithms.

9 Related Work

Relevant work on top-k query processing can roughly be divided in two groups: evaluation strategies for multiattribute top-k queries over multimedia repositories, and for top-k queries over relational databases.

To process queries involving multiple multimedia attributes, Fagin et al. proposed a family of algorithms [9, 10], developed as part of IBM Almaden's Garlic project. These algorithms can evaluate top-k queries that involve several independent multimedia "subsystems," each producing scores that are combined using arbitrary monotonic aggregation functions. In an expanded version of [10], Fagin et al. presented a variation of their algorithms to handle *R-Sources*. We experimentally compared Fagin et al.'s algorithms with our new approach in Sections 7 and 8.

Nepal and Ramakrishna [18] and Güntzer et al. [12] presented variations of Fagin's original FA algorithm [9] for processing queries over multimedia databases. In particular, Güntzer et al. [12] reduce the number of random accesses through the introduction of more stop-condition tests and by exploiting the data distribution. The MARS system [19] also uses variations of the FA algorithm and views queries as binary trees where the leaves are single-attribute queries and the internal nodes correspond to "fuzzy" query operators. More recently, Chang and Hwang [5] presented *MPro*, an algorithm that is closely related to *Upper*. Unlike *Upper*, *MPro* assumes a fixed schedule of accesses to *R-Sources*, and thus selects which object to probe next but ignores source selection on a per-object level. Therefore, we can consider *MPro* as a special case of the *Upper* algorithm in which the *SelectBestSource* function is fixed and always returns the same sequence of sources. We experimentally compared *MPro* with *Upper* in Section 7.2.

Chaudhuri and Gravano also built on Fagin's original FA algorithm and proposed a cost-based approach for optimizing the execution of top-k queries over multimedia repositories [6]. Their strategy translates a given top-k query into a selection query that returns a (hopefully tight) superset of the actual top-k tuples. Ultimately, the evaluation strategy consists of retrieving the top-k' tuples from as few sources as possible, for some $k' \geq k$, and then probing the remaining sources by invoking existing strategies for processing selections with expensive predicates [13, 15]. This technique is then related to algorithm *TA-EP* from Section 3.3.2.

Over relational databases, Carey and Kossmann [3, 4] presented techniques to optimize top-k queries when the scoring is done through a traditional SQL order-by clause. Donjerkovic and Ramakrishnan [8] proposed a probabilistic approach to top-k query optimization. Chaudhuri and Gravano [1] exploited multidimensional histograms to process top-k queries over an unmodified relational DBMS by mapping top-k queries into traditional selection queries. Finally, Chen and Ling [7] used a sampling-based approach to translate top-k queries over relational data into approximate range queries.

Additional related work includes the PREFER system [14], which uses pre-materialized views to efficiently answer ranked preference queries over commercial DBMSs. Recently, Natsev et al. proposed incremental algorithms [17] to compute top-k queries with user-defined join predicates over sorted-access sources. Finally, the WSQ/DSQ project [11] presented an architecture for integrating web-accessible search engines with relational DBMSs. The resulting query plans can manage asynchronous external calls to reduce the impact of potentially long latencies. The WSQ/DSQ ideas could be incorporated to speed up the execution of our top-k queries further and depart from the sequential query plans on which we focused in this paper.

10 Conclusion

In this paper, we studied the problem of processing top-k queries over autonomous web-accessible sources with a variety of access interfaces. We first focused on a scenario with multiple random-access sources and one sorted-access source. We adapted and improved existing algorithms for top-k query processing to our scenario, and also introduced a novel strategy, *Upper*, which is designed specifically for our query model. A distinctive characteristic of our new algorithm is that it interleaves probes on several objects whereas other techniques completely probe one object at a time. This interleaving has a strong effect on query processing efficiency. We analyzed the space and time requirements of our various techniques and described data structures to implement them efficiently. We conducted a thorough experimental evaluation of these techniques using both synthetic and real web-accessible data sets. Our evaluation showed that *Upper* produces the best processing plans in terms of execution time for a variety of data and query parameters, and for both synthetic and real data sets. Finally, we relaxed our source model to handle any number of random-access sources and of sources supporting both sorted and random access. We adapted our *Upper* technique to this relaxed model and experimentally compared it to an existing algorithm, TA_z [10]. Our results showed that adaptations of *Upper* perform significantly better, confirming that interleaving probes on objects improves query processing efficiency.

References

[1] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *To appear in ACM Transactions on Database Systems*, 2002.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proc. of the 2002 International Conference on Data Engineering (ICDE'02)*, 2002.

[3] M. J. Carey and D. Kossmann. On saying “Enough Already!” in SQL. In *Proc. of the 1997 ACM International Conference on Management of Data (SIGMOD'97)*, May 1997.

[4] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the Twenty-fourth International Conference on Very Large Databases (VLDB'98)*, Aug. 1998.

[5] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proc. of the 2002 ACM International Conference on Management of Data (SIGMOD'02)*, 2002.

[6] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proc. of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, 1996.

[7] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In *Proc. of the 2002 International Conference on Data Engineering (ICDE'02)*, 2002.

[8] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *Proc. of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, 1999.

[9] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of the Fifteenth ACM Symposium on Principles of Database Systems*, 1996.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the Twentieth ACM Symposium on Principles of Database Systems*, 2001. Expanded version appears on `http://www.almaden.ibm.com/cs/people/fagin/`.

[11] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proc. of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*, 2000.

[12] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *Proc. of the Twenty-sixth International Conference on Very Large Databases (VLDB'00)*, 2000.

[13] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the 1993 ACM International Conference on Management of Data (SIGMOD'93)*, 1993.

[14] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proc. of the 2001 ACM International Conference on Management of Data (SIGMOD'01)*, 2001.

[15] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of the 1994 ACM International Conference on Management of Data (SIGMOD'94)*, 1994.

[16] D. Mills. Internet delay experiments; RFC 889. In *ARPANET Working Group Requests for Comments*, number 889. SRI International, Menlo Park, CA, Dec. 1983.

[17] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *Proc. of the Twenty-seventh International Conference on Very Large Databases (VLDB'01)*, 2001.

[18] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proc. of the 15th International Conference on Data Engineering*, 1999.

[19] M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T. S. Huang. Supporting ranked boolean similarity queries in MARS. *TKDE*, 10(6):905–925, 1998.

[20] S. A. Williams, H. Press, B. P. Flannery, and W. T. Vetterling. *Numerical Recipes in C: The art of scientific computing*. Cambridge University Press, 1993.