Evaluating Top-k Queries over Web-Accessible Databases

Nicolas Bruno Luis Gravano Amélie Marian
Computer Science Department
Columbia University
{nicolas,gravano,amelie}@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 algorithms for processing such queries, and evaluate them thoroughly using both synthetic and real web-accessible data.

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.

[1] http://www.zagat.com
[2] http://www.nytoday.com
[3] http://www.mapquest.com

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

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.

Recently, Fagin et al. [7] 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). Unfortunately, these algorithms are not designed for sources that only support random access (e.g., the MapQuest site), which abound on the web. In fact, as we will see, simple adaptations of these algorithms do not perform well over random-access sources. In this paper, we present novel processing strategies for top-k queries over sources that support just random access, just sorted access, or both. We also develop non-trivial adaptations of Fagin et al.'s algorithms for random-access sources, 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 model, notation and terminology that we use in Section 3 to present our new techniques and our adaptations of Fagin et al.'s algorithms. We evaluate the different strategies experimentally in Section 5 using the data sets and metrics in Section 4. Section 6 reviews relevant work.

2. Query Model

In traditional relational systems, query results consist of a 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: [4]

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

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.

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 [4, 5] 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 [5].

[4] 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.

[5] 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.

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 $\texttt{getNext}_S(q_i)$ 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 $\texttt{getScore}_R(q_i, t)$ 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:

Definition 2 : [Access Cost] *Consider an R-Source or SR-Source R 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 an S-Source or SR-Source S. 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. [7] presented "instance optimal" query processing algorithms over sources that are either of type *SR-Source* (TA algorithm) or of type *S-Source* (NRA algorithm). As we will see, simple adaptations of these algorithms do not perform as well for the common scenario where *R-Source* sources are also available. In the remainder of this paper, we address this limitation of existing top-k query processing techniques.

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 for similar problems to our framework.

We make a number of simplifying assumptions in the remainder of this section. Specifically, we assume that the scoring function for all attributes return values 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. (See Section 7.)

Following Fagin et al. [6, 7], 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-Source* $R_1, \ldots, R_n$ "know about" all objects in $Objects(S)$. In other words, given a query q and an object $t \in Objects(S)$, we can probe R_i and obtain the score $Score_{A_i}(q_i, t)$ 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 $Score_{A_i}(q_i, t)$.

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 $Objects(S)$, 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 $Objects(S)$. 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 $|Objects(S)| \cdot (tS(S) + \sum_{i=1}^{n} tR(R_i))$.

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 cost 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 an object t that has been retrieved from *S-Source* S and for which we have already probed some subset of *R-Sources* $R' \subseteq \{R_1, \ldots, R_n\}$. 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 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: $L(t) = \mathit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n)$, 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 $e(s_i)$: $E(t) = \mathit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n)$, where $\hat{s}_i = s_i$ if $R_i \in R'$, and $\hat{s}_i = e(s_i)$ otherwise. If object t has not been retrieved from S yet, then we define $E(t) = \mathit{ScoreComb}(e(s_0), e(s_1), \ldots, e(s_n))$. In the absence of more sophisticated statistics we set the expected partial score $e(s_i)$ to 0.5 for $i = 1, ..., n$, and $e(s_0)$ to $\frac{s_\ell}{2}$, 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 [6]. ($\mathit{Score}_{A_0}(q_0, t_0)$ can range between 0 and s_ℓ.)

In Section 3.2.1 we define what constitutes an optimal 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. Finally, in Section 3.2.3 we derive another technique, *Pick*, which, at each step, aims at greedily minimizing some "distance" between the current execution state and the final state.

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

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 cost 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 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 upper bound $U(t)$ for an object $t \in$ Objects(S) is strictly lower than the 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 upper bound 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's 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.

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 [7]. (*Optimal* assumes complete knowledge of all object scores.)
2. Let $score_k$ be the lowest score in $Answer_k$.
3. Get the best object t for attribute A_0, with score s_0, from *S-Source* S: $(t, s_0) \leftarrow \mathtt{getNext}_S(q_0)$.
4. If $U(t) \leq score_k$ and we have seen all objects in $Answer_k$, return the top-k objects after completely probing them and stop. (No unretrieved object in $\mathit{Objects}(S)$ can have a higher upper bound than t.)

[7] In the presence of score ties, to ensure optimality, this step picks the objects that would be the most expensive to discard in Step 5(b).

5. (a) If object t is one of the $Answer_k$ objects, probe all *R-Sources* to compute *Score*(q, t).
   (b) 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 cost $\sum_{R_i \in R'} tR(R_i)$ is minimal among the subsets of $\{R_1, \ldots, R_n\}$ with the property above.
6. Get the next best object t from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$ and return to step 4.

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 most likely 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 $t \in$ Objects(S) has a score upper bound $U(t)$ strictly higher than that of every other object (i.e., $U(t) > U(t')\ \forall t' \neq t \in$ 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 of the actual top-k objects, 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, its upper bound $U(t)$ is higher than the score of any of the top-k objects. Hence t requires further probes so that $U(t)$ decreases before a final answer can be established.*

This property is illustrated in Figure 1 for a top-3 query. In this figure, each object's possible range of scores is represented by a segment, and objects are sorted by their expected score. From Property 1, objects 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.) Also, from Property 2, the object with the highest 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 value needs to be returned, or its upper bound will have to be lowered through further probes so that we can safely discard it. In practice, when several objects agree on the highest upper bound, one of them will be arbitrarily chosen for the next probe.

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. Get the best object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
2. Initialize $U_{unseen} = U(t)$, $Candidates = \{t\}$, and $returned = 0$.
3. If $Candidates \neq \emptyset$:
   - Pick t_H from $Candidates$ such that $U(t_H) = \max_{t' \in Candidates} U(t')$.

   Else: t_H is undefined.
4. If t_H is undefined or $U(t_H) < U_{unseen}$:
   - Get the next best 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:
   - Return t_H with its score; remove t_H from $Candidates$.
   - $returned = returned + 1$. If $returned = k$, halt.

     Else:
   - $Ri \leftarrow SelectBestSource(t_H, Candidates, k - returned)$.
   - Probe source R_i on object t_H: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t_H)$.
5. Go to step *3*.

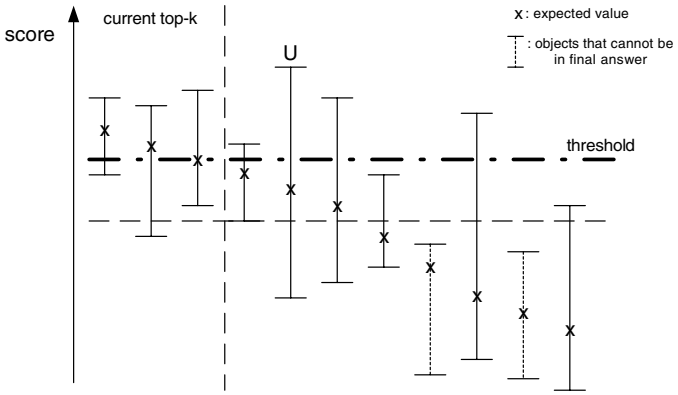


Figure 1. Snapshot of the execution of the *Upper* strategy.

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 in step 4 of the algorithm. As in *Optimal*, we concentrate on (a) computing the final value of the top-k objects, and (b) for all other objects, decreasing their upper bound so that it does not exceed the scores of the top-k objects. However, unlike *Optimal*, *Upper* does not know the actual scores a-priori and must rely on expected values to make its choices.

For an object t, we select the best source to probe as follows. If t is expected to be in the final answer, i.e., its expected value is one of the k highest ones, we compute its final score, and all sources not yet probed for t are considered. Otherwise, we only consider the fastest subset of sources not probed for t that is expected to decrease $U(t)$ to not exceed the value of the k^{th} largest expected score (*threshold* T). The best source for t is the one that has the highest $\frac{weight}{cost}$ ratio, i.e., the one that is expected to have a high impact on t's possible score range while being fast:

Function SelectBestSource (Input: object t, set of objects Candidates, integer r)

1. Let t' be the object in $Candidates$ with the r^{th} largest expected score. Let $T = E(t')$.
   (a) If $E(t) \geq T$:
   - Define $R' \subseteq \{R_1, \ldots, R_n\}$ as the set of all sources not yet probed for t.
     (t is expected to be one of the top-k objects, so it needs to be probed on all attributes.)

   (b) Else, define $R' \subseteq \{R_1, \ldots, R_n\}$ so that:
   - $U(t) \leq T$ if each source $R_i \in R'$ were to return the expected value for t, and
   - The cost $\sum_{R_i \in R'} tR(R_i)$ is minimal among the subsets of $\{R_1, \ldots, R_n\}$ with the above property.

   (Since $E(t) < T$, we are guaranteed to find at least one such set of attributes.)
2. Return a source $R_i \in R'$ such that $\frac{w_i}{tR(R_i)}$ is maximum.

3.2.3 The Pick Strategy

We now present the *Pick* algorithm, which uses an alternative approach to evaluate top-k queries. While *Upper* chooses the probe that is most likely to be in the "optimal" set of probes, at each step *Pick* chooses the probe that minimizes a certain function B, which represents the "distance" between the current execution state and the final state, in which the top-k tuples are easily extracted. At a given point in time in the execution, function B focuses on t', the object with the k^{th} highest expected score among the objects retrieved from *S-Source* S. Furthermore, B considers the range of possible scores that each such object can take above $E(t')$. The smaller such ranges are, the closer we are to finding the final solution for the query. In effect, when we reach the final state, t' is the object with the actual k^{th} highest score, and all objects not in the answer should be known *not* to have scores above that of t'. The definition of function B is:[8]

$$B = \sum_{t \in \textit{Objects}(S)} \max\{0,\ U(t) - \max\{L(t), E(t')\}\}$$

Figure 2 shows a snapshot of a query execution step, highlighting the score ranges that "prevent" the current state from being the final state. Note that the value of B is never negative. When B becomes zero, all top-k scores are known, and all objects not in the final answer have an upper bound for their score that is no higher than $E(t')$.

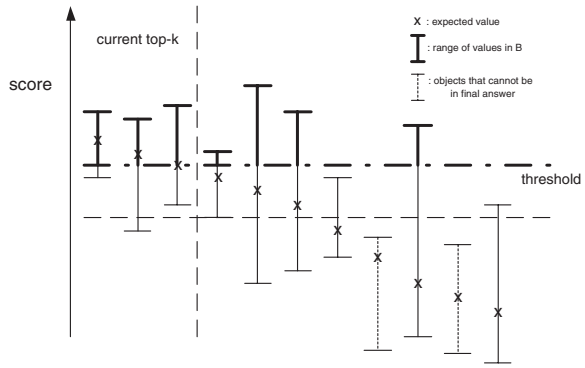


Figure 2. Snapshot of the execution of the *Pick* strategy.

At each step, *Pick* greedily chooses the probe that would decrease B the most in the shortest time. *Pick* selects for each object the best source to probe, i.e., the attribute value that will result in the highest decrease in B. If the object is expected to be in the final top-k answer, all unprobed attributes are considered. Otherwise, only attributes from the best (fastest) set of attributes that will be needed to eliminate the object are considered. This is completely analogous to how the *SelectBestSource* function works. Then, among all selected object-*R-Source* pairs, *Pick* chooses the one with the highest $\frac{expected\ decrease\ of\ B}{tR(R)}$ ratio.

Observe that, unlike *Upper*, *Pick* retrieves all candidate objects to consider during an initialization step, and does not access the *S-Source* afterwards.

[8] We studied several alternative definitions for B that did not work as well in our experiments as the one that we present here. For space limitations we do not discuss these alternatives further.

Algorithm Pick (Input: top-k query q)

1. Retrieve all objects that can be in the top-k solution from *S-Source* S:
   (a) Get the k best objects $t_1, ..., t_k$ for attribute A_0 from *S-Source* S: $(t_i, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
   (b) Initialize $Candidates = \{t_1, ..., t_k\}$; initialize $t = t_k$.
   (c) While $L(t_k) < U(t)$, get the next best object t for attribute A_0 from S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$; insert t into $Candidates$.
2. While $B > 0$:
   (a) For each object $t \in Candidates$ select the best source: $R_i \leftarrow SelectBestSource(t, Candidates, k)$.
   (b) Choose among the selected pairs (t, R_i) the one that has the highest expected gain per unit of time ($\frac{expected\ decrease\ in\ B}{tR(R)}$) and probe it.
3. Return the top-k objects.

Selecting a probe using *Pick* is more expensive than with *Upper* since we have to consider probes on all objects. Moreover, *Pick* needs to retrieve all objects that might belong to the top-k answer from the *S-Source* at initialization, which might result in all objects being retrieved.

3.3. Existing Approaches

Existing algorithms in the literature assume that all sources are *SR-Sources* or *S-Sources*, and do not directly handle sources with only a random-access interface. In Section 3.3.1 we adapt Fagin et al.'s TA algorithm [7] so that it also works over *R-Sources*, and in Section 3.3.2 we extend the resulting algorithm so that it also incorporates ideas from the expensive-predicates literature. 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 5.

3.3.1 Fagin et al.'s Algorithms

Fagin et al. [7] presents 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 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 value of any object that has not been seen so far in any source.
3. If the current top-k objects seen so far have scores greater than or equal to the threshold, return those values. 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, and retrieve an object t. In step 2, we define the threshold value as $U(t)$, since the maximum possible score for any *R-Source* is always 1. Then, for each object t retrieved from S we probe all *R-Sources* to get the final score for t. For a model with a single *S-Source* S, the modified algorithm retrieves in order all objects in *Objects*(S) one by one and determines whether each object is in the final answer by probing the remaining *R-Sources*. The complete procedure is described next.

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

1. Get the best object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
2. Update threshold $T = U(t)$.
3. Retrieve score s_i for attribute A_i and object t via a random probe to *R-Source* R_i: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t)$ for $i = 1, \ldots, n$.
4. Calculate t's final score for q: $score = ScoreComb(s_0, s_1, \ldots, s_n)$.
5. If $score$ is one of the top-k scores seen so far, keep object t along with its score.
6. (a) If threshold T is lower than or equal to the scores of the current k top objects, return these k objects along with their scores and stop.
   (b) Otherwise, get the next best object t from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$ and return to step 2.

We can improve the algorithm above by interleaving the execution of steps 3 and 4 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 step 5 if the current object t is guaranteed not to be a top-k object. 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*.

3.3.2 Exploiting Techniques for Processing Selections with Expensive Predicates

Work on expensive-predicate query optimization [10, 12] 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 $cost\text{-}per\text{-}object(p_i)$ is the average time to evaluate predicate 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. We sort the *R-Sources* R_i in decreasing order of rank, defined as: $rank_{R_i} = \frac{w_i \cdot (1-E(Score_{R_i}))}{tR(R_i)}$, where $E(Score_{R_i})$ is the expected score of an object from source R_i (typically 0.5 unless we have additional information). 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 $U(t)$.

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. Get the best object t for attribute A_0 from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
2. Update threshold $T = U(t)$.
3. For each *R-Source* R_i in decreasing order of $rank_{R_i}$:
   (a) Retrieve score s_i for attribute A_i and object t via a random probe to *R-Source* R_i: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t)$.
   (b) If $U(t)$ is lower than or equal to the score of k objects, skip to step 4.
4. If t's score is one of the top-k scores seen so far, keep object t along with its score.
5. (a) If threshold T is lower than or equal to the scores of the current k top objects, return these k objects along with their scores and stop.
   (b) Otherwise, get the next best object t from *S-Source* S: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$ and return to step 2.

4. Evaluation Setting

In this section we describe the data sets (Section 4.1), metrics and other settings (Section 4.2) that we use to evaluate the strategies of Section 3.

4.1. Data Sets

Synthetic Sources: We generate different synthetic data sets. Objects in these data sets have attributes from a single *S-Source* S and five *R-Sources*. The data sets vary in their 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 [16].

The random-access cost 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 cost for *S-Source* S (i.e., $tS(S)$) is randomly picked from $\{0.1, 0.2, \ldots, 1.0\}$.

Real Web-Accessible Sources: The real sources that we use are relevant to (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 [9]), *DrivingTime* (handled by the MapQuest site), *Popularity* (handled by the AltaVista search engine [10]; see below), *ZFood*, *ZPrice*, *ZDecor*, and *ZService* (handled by the Zagat Review web site), and *TRating* and *TPrice* (provided by the New York Times at the New York Today web site). The Verizon Yellow Pages listing [11], which returns restaurants of the user-specified type sorted by shortest distance from a given address, is the only *S-Source*. 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 [8].) Table 1 summarizes these sources and their interfaces.

Of course, the real sources above do not fit our model of Section 2 perfectly. For example, some of these sources return values 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) value to the score of the corresponding attribute.

[9] http://www.subwaynavigator.com
[10] http://www.altavista.com
[11] http://www.superpages.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.

4.2. Other Experimental Settings

Our query processing strategies attempt to minimize the total processing time for top-k queries, both for random and sorted access to the various 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:

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

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_{total} then approximates the execution time for a query.

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_{total} values. We report results for top-k queries for different values of k, $|S|$, cf and for various assignments of weights and costs to sources. In the default setting, k is 50 (i.e., queries ask for the best 50 objects), $|S| = 10,000$, and we use the *Uniform* data set.

For the real data sets, we use seven queries, some specifying an address on E. 73^{rd} Street, and some others specifying an address on W. 112^{th} Street. Attributes *Distance*, *SubwayTime*, *DrivingTime*, *ZFood*, *ZDecor*, *ZService*, 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 1,000 hits, while *ZPrice* and *TPrice* are set to the least expensive value in the scale. In all seven queries, the weight of the *S-Source* attribute (i.e., *Distance*) is roughly twice the weight of any *R-Source* attribute.

Next, we experimentally compare the algorithms that we discussed in Section 3, namely *TA-Adapt* (Section 3.3.1), *TA-Opt* (Section 3.3.1), *TA-EP* (Section 3.3.2), and our *Upper* (Section 3.2.2) strategy. 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. We do not present results for *Pick* (Section 3.2.3) for lack of space. In our experiments, *Pick* performed on average slightly worse than *Upper* in terms of source access costs, but with significantly higher "local" processing time. (Recall from Section 3.2.3 that *Pick* selects the best source to probe for every object at each iteration of the algorithm.)

5. Evaluation Results

In this section we present the experimental results for the techniques of Section 3 using the data sets and general settings described in Section 4.

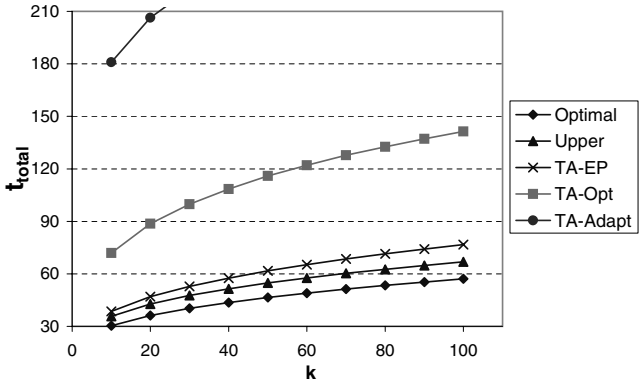
5.1. Results for Synthetic Data Sets

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

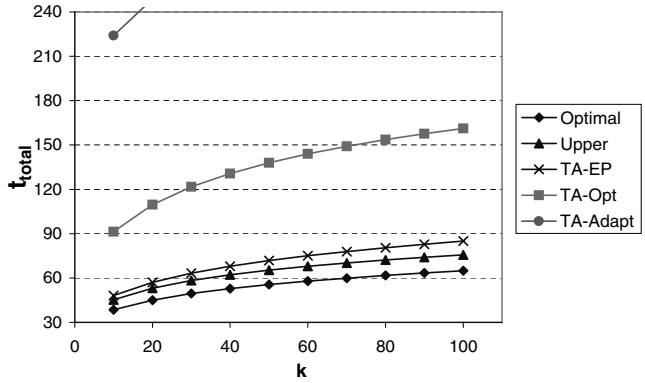
Effect of the Number of Objects Requested k: In Figure 3 we report results for the default setting, as a function of k and for both the *Uniform* and *Gaussian* synthetic data sets. As k increases, the time needed by each algorithm to return the top-k objects increases as well, since all techniques need to retrieve and process more objects. The *Upper* strategy consistently outperforms all other techniques, and has total execution time close to that of the 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*. We then remove *TA-Adapt* from further consideration in the remaining discussion.

Effect of the Number of Objects in *S-Source* S: Figure 4 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 time needed by each algorithm is approximately linear in the number of objects in S. *Upper* gives better results and scales better than other techniques.

Effect of Attribute Weights: We now report on the impact of attribute weights on the execution times. We vary the weight of the *S-Source* S (Figure 5(a)) and the *R-Source* R_5 (Figure 5(b)) relative to the weight of the remaining sources. In particular, we set the varying weight as n times the average of the remaining weights. Figure 5(a) shows that for larger weights in *S-Source* S all techniques improve their execution times, since fewer random probes are needed to identify the top-k objects. Also, for larger weights in *R-Source* R_5 (Figure 5(b)), *TA-Opt* performs poorly since it does not use any information about the relative weights of the sources to order the random probes, in contrast to the other techniques. We



(a) *Uniform data set*



(b) *Gaussian data set*

Figure 3. 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.

note that we obtained the same results when we varied the access costs of the different sources instead of their weights.

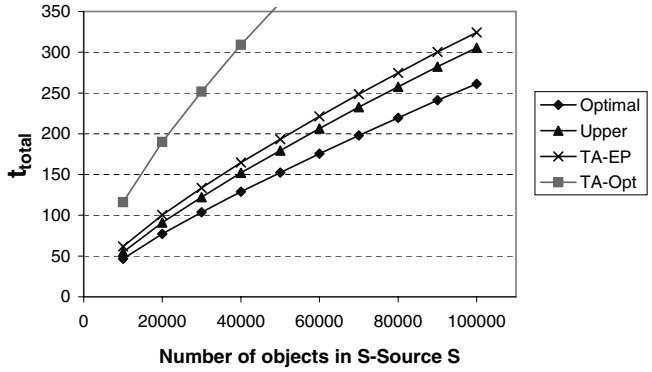


Figure 4. Performance of the different strategies for the *Uniform* data set, as a function of the number of objects in *S-Source* S.

Effect of Attribute Correlation: We now turn to the *Correlation* data set (Section 4) and evaluate the effect that attribute correlation has on the performance of the query processing techniques. As seen in Figure 6, 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* 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 the tuples in *S-Source* S before returning the top-k objects, which results in significantly larger execution times compared to *Upper*.

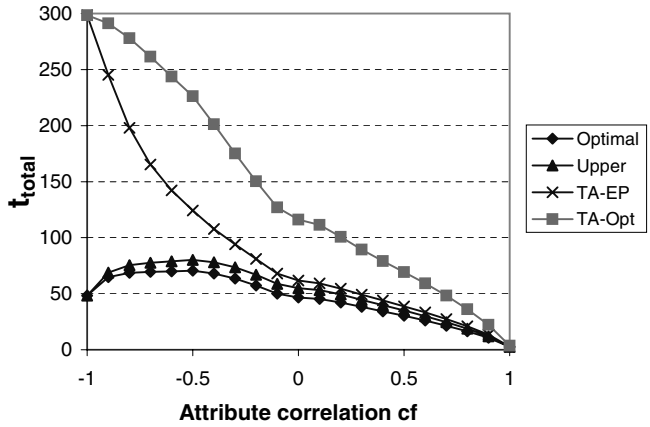
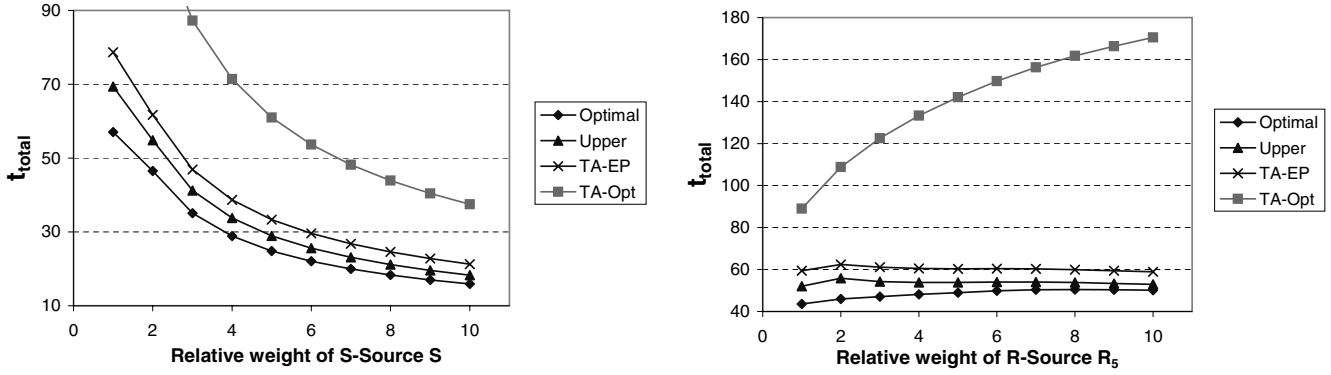


Figure 6. Performance of the different strategies for the *Correlation* synthetic data set, as a function of the correlation factor *cf*.

5.2. Results for Real Web-Accessible Data Sets

Our final set of results are for the real data sets that we described in Section 4 and summarized in Table 1. There are six web-accessible sources, handling 10 attributes. To model the access cost for each source, we measured the response time for a number of queries and computed their average. We then issued seven different queries to these sources and timed their execution. Figure 7 shows the execution time for each of the queries, and for the *Upper*, *TA-EP*, and *TA-Opt* strategies. We ignored *TA-Adapt*, whose results in the synthetic data experiments were significantly worse than those for other techniques. In contrast with the synthetic-data results, *TA-EP* does not outperform *TA-Opt*. We conjecture



(a) Varying the relative weight of the S-Source

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

Figure 5. Performance of the different strategies for various attribute-weight combinations.

that this discrepancy is due to our rough estimates for the source access costs, to which the *TA-EP* strategy would be particularly sensitive. In general, just as for the synthetic data sets, our *Upper* strategy performs significantly better than the two versions of the *TA* algorithm.

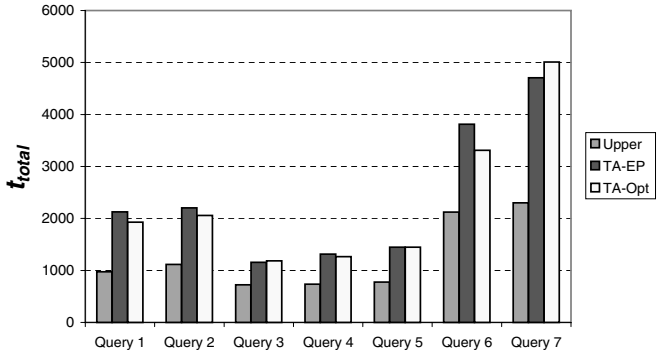


Figure 7. Experimental results for the real web-accessible data sets relevant to our New York City restaurant scenario.

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 to the *TA* 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. The design of appropriate scoring functions is an important problem that we do not address in this paper.

6. 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 [6, 7], 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. These techniques do not directly handle sources that provide only a random-access interface, which are the focus of our paper. In Section 3.3, however, we adapted Fagin et al.'s algorithms to our scenario and experimentally compared the resulting techniques with our new approach in Section 5.

Nepal and Ramakrishna [14] and Güntzer et al. [9] presented variations of Fagin's original FA algorithm [6] for processing queries over multimedia databases. In particular, Güntzer et al. [9] reduce the number of random accesses through the introduction of more stop-condition tests and by exploiting the data distribution. The MARS system [15] 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.

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 [3]. 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 [10, 12]. This technique is then closely related to algorithm TA-EP from Section 3.3.2.

Over relational databases, Carey and Kossmann [1, 2] presented techniques to optimize top-k queries when the scoring is done through a traditional SQL order-by clause. Donjerkovic and Ramakrishnan [5] proposed a probabilistic ap-

proach to top-k query optimization. Finally, Chaudhuri and Gravano [4] exploited multidimensional histograms to process top-k queries over an unmodified relational DBMS by mapping top-k queries into traditional selection queries.

Additional related work includes the PREFER system [11], which uses pre-materialized views to efficiently answer ranked preference queries over commercial DBMSs. Recently, Natsev et al. proposed incremental algorithms [13] to compute top-k queries with user-defined join predicates over sorted-access sources. Finally, the WSQ/DSQ project [8] 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.

7. Conclusion and Future Work

We studied techniques to efficiently evaluate top-k queries over web-accessible autonomous databases with a variety of access interfaces. In particular, we focused on web sources that can only be accessed via random accesses. We proposed extensions to existing algorithms for top-k queries so that they can handle random-access sources, and also introduced two novel strategies, *Upper* and *Pick*, which are designed specifically for our query model. A distinctive characteristic of our algorithms is that they interleave 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 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.

We plan to investigate several interesting directions in the future. Our model assumes that only one *S-Source* and multiple *R-Sources* are available; we are evaluating algorithms that would work in the general case with arbitrary numbers of *S-Sources*, *R-Sources*, and *SR-Sources*. (At least one *S-Source* or *SR-Source* is always needed, though, not to rely on "wild guesses.") Another assumption is that only one source can be accessed at a time, which is too restrictive in the context of web sources. As explained in Section 6, we can incorporate the ideas in [8] to include parallelism and speed up query processing.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grants No. IIS-97-33880 and IIS-98-17434, and by a gift from Microsoft Research. Amélie Marian was partially supported by a Viros Scholarship. We also thank Ron Fagin for helpful comments on an earlier version of this paper.

References

[1] 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.

[2] 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.

[3] 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.

[4] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proc. of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, 1999.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.