Evaluating Top-*k* Queries Over Web-Accessible Databases

AMÉLIE MARIAN
Columbia University, New York
NICOLAS BRUNO
Microsoft Research, Redmond, Washington
and
LUIS GRAVANO
Columbia University, New York

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 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 article, 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 a sequential algorithm for processing such queries, but observe that any sequential top-k query processing strategy is bound to require unnecessarily long query processing times, since web accesses exhibit high and variable latency. Fortunately, web sources can be probed in parallel, and each source can typically process concurrent requests, although sources may impose some restrictions on the type and number of probes that they are willing to accept. We adapt our sequential query processing technique and introduce an efficient algorithm that maximizes source-access parallelism to minimize query response time, while satisfying source-access constraints.

This material is based upon work supported in part by the National Science Foundation under grants IIS-97-33880 and IIS-98-17434, as well as by a gift from Microsoft Research.
A preliminary version of this article appeared in *Proceedings of the 2002 International Conference on Data Engineering (ICDE'02)* as Bruno et al. [2002b].
Authors' addresses: A. Marian and L. Gravano, Computer Science Department, Columbia University, 1214 Amsterdam Ave., New York, NY 10027; email: {amelie,gravano}@cs.columbia.edu; N. Bruno, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: nicolasb@microsoft.com.

We evaluate our techniques experimentally using both synthetic and real web-accessible data and show that parallel algorithms can be significantly more efficient than their sequential counterparts.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation (efficiency and effectiveness)*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Parallel query processing, query optimization, top-*k* query processing, 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 "structured" data for applications where users do not expect exact answers to their queries, but instead a rank of the objects that best match the 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 website,[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 website.[2] Finally, the scoring associated

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

with the *Address* attribute might be handled by the MapQuest website,[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.

Our query scenario is related to a (centralized) multimedia query scenario where attributes are reached through several independent multimedia "subsystems," each producing scores that are combined to compute a top-*k* query answer. While multimedia systems might support sorted and random attribute access, there are important differences between processing top-*k* queries over multimedia systems and over web sources. First, web sources might only support random access (e.g., MapQuest returns the distance between two given addresses). Second, attribute access for centralized multimedia systems might be faster than for web sources, because accessing web sources requires going over the Internet. Finally, and importantly, unlike in multimedia systems where attribute access requires "local" processing, applications accessing web sources can take full advantage of the intrinsic parallel nature of the web and issue probes to several web sources simultaneously, possibly issuing several concurrent probes to each individual source as well.

In this article, we present algorithms to efficiently process top-*k* queries over web sources that support just random access and, optionally, sorted access as well. We first introduce an efficient sequential top-*k* query processing algorithm that interleaves sorted and random accesses during query processing and schedules random accesses at a fine-granularity per-object level. Then, we use our sequential technique as the basis to define a parallel query processing algorithm that exploits the inherently parallel nature of web sources to minimize query response time. As we will see, making the algorithms parallel results in drastic reductions in query processing time.

The rest of the article is structured as follows. Section 2 defines our query and data models, notation and terminology. Section 3 reviews relevant work. Section 4 presents our sequential top-*k* query processing technique, as well as improvements of algorithms presented by Fagin et al. [2001] that are applicable to our scenario. Section 5 focuses on parallel query-processing strategies, and presents an algorithm that fully exploits the available source-access parallelism, resulting in reduced query processing times. We evaluate the different

[3]`http://www.mapquest.com.`

strategies experimentally in Section 7 using the data sets and metrics presented in Section 6. Finally, Section 8 concludes the paper.

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 closely 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 $A_1, \ldots, A_n$, plus perhaps some other attributes not mentioned in our queries. A top-k query over relation R simply specifies target values for each attribute A_i. Therefore, a top-k query is an assignment of values $\{A_1 = q_1, \ldots, A_n = q_n\}$ 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 a top-k query $q = \{A_1 = q_1, \ldots, A_n = q_n\}$ 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. We assume that the scoring function for each individual attribute returns scores between 0 and 1, with 1 denoting a perfect match. 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]:

$$Score(q, t) = ScoreComb(s_1, \ldots, s_n) = \sum_{i=1}^{n} w_i \cdot s_i$$

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. In this article, we only consider techniques that return the top-k objects *along with their scores*.

Example 1 (cont.). We can define the scoring function for the *Address* attribute of a query and an object as inversely proportional to 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 based on the object's value for this attribute. If price

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

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*.

This article 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. For instance, the *Price* attribute in our example is provided by the NYT-Review website and can only be accessed by querying this site's web interface.[5] We distinguish between three types of sources based on their access interface:

Definition 2.1 (*Source Types*). Consider an attribute A_i and 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 we can obtain from S a list of objects sorted in descending order of $Score_{A_i}$ by (repeated) invocation of a `getNext`(S, q) 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 `getScore`(R, q, 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 website, which returns the distance between the restaurant address and the user-specified address. Hence, MapQuest is an *R-Source*.

At a given point in time during the evaluation of a top-*k* query q, we might have partial score information for an object, after having obtained some of the object's attribute scores via source probes, but not others:

—$U(t)$, the *score upper bound* for an object t, is the maximum score that t might reach for q, consistent with the information already available for t. $U(t)$ is then the score that t would get for q if t had the maximum possible score for every attribute A_i not yet probed for t. In addition, we define U_{unseen} as the score upper bound of any object not yet retrieved from any source via sorted accesses.

—$E(t)$, the *expected score* of an object t, is the score that t would get for q if t had the "expected" score for every attribute A_i not yet probed for t. In absence of further information, the expected score for A_i is assumed to be 0.5 if its associated source D_i is an *R-Source*, or $\frac{s_\ell(i)}{2}$ if D_i supports sorted

[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.

accesses, where $s_\ell(i)$ is the $Score_{A_i}$ score of the last object retrieved from D_i via sorted access. (Initially, $s_\ell(i) = 1$.) Several techniques can be used for estimating score distribution (e.g., sampling); we will address this issue in Sections 7.1.3 and 7.2.2.

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 2.2 (*Access Costs*). 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 $\boldsymbol{tR(R)}$. (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 for the associated attribute as $\boldsymbol{tS(S)}$. (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.

We make a number of assumptions in our presentation. The top-k evaluation strategies that we consider do not allow for "wild guesses" [Fagin et al. 2001]: an object must be "discovered" under sorted access before it can be probed using random access. Therefore, we need to have at least one source with sorted access capabilities to discover new objects. We consider n_{sr} *SR-Sources* $D_1, \ldots, D_{n_{sr}}$ ($n_{sr} \geq 1$) and n_r *R-Sources* $D_{n_{sr}+1}, \ldots, D_n$ ($n_r \geq 0$), where $n = n_{sr} + n_r$ is the total number of 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 scores for the top-k objects. This can be extremely expensive in practice. Fagin et al. [2001] presented the *NRA* algorithm to deal with multiple *S-Sources*; however, *NRA* only identifies the top-k objects and does not compute their final scores. We plan to relax this restriction and adapt our algorithms to handle *S-Sources* in addition to *SR-Sources* and *R-Sources* in future work.

We refer to the set of all objects available through the sources as the *Objects* set. Additionally, we assume that all sources $D_1, \ldots, D_n$ "know about" all objects in *Objects*. In other words, given a query q and an object $t \in Objects$, we can obtain the score 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 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 simply use a default value for t's score for attribute A_i.

3. RELATED WORK

To process queries involving multiple multimedia attributes, Fagin et al. proposed a family of algorithms [Fagin 1996; Fagin et al. 2001, 2003] 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. The initial *FA* algorithm [Fagin 1996] was followed by "instance optimal" query processing algorithms over sources that are either of type *SR-Source* (*TA* algorithm) or of type *S-Source* (*NRA* algorithm) [Fagin et al. 2001]. In later work, Fagin et al. [2003] 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. We discuss these algorithms in Section 4.1, and show how they can be adapted to our parallel access model in Section 5.2. We also compare them experimentally against our techniques in Section 7.

Nepal and Ramakrishna [1999] and Güntzer et al. [2000] presented variations of Fagin et al.'s *TA* algorithm [Fagin et al. 2001] for processing queries over multimedia databases. In particular, Güntzer et al. [2000] reduce the number of random accesses through the introduction of more stop-condition tests and by exploiting the data distribution. The MARS system [Ortega et al. 1998] 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 et al. built on Fagin's original *FA* algorithm and proposed a cost-based approach for optimizing the execution of top-*k* queries over multimedia repositories [Chaudhuri and Gravano 1996; Chaudhuri et al. 2004]. 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 [Hellerstein and Stonebraker 1993; Kemper et al. 1994]. This technique is thus related to the TA_z-*EP* algorithm that we present in Section 4.1.2.

More recently, Chang and Hwang [2002] presented *MPro*, an algorithm to optimize the execution of *expensive predicates* for top-*k* queries, rather than for our web-source scenario. *MPro* is more general than our techniques in that it targets a wider range of scenarios: local expensive predicates, external expensive predicates, arbitrary monotonic scoring functions, and joins. Their "probes" are typically not as expensive as our web-source accesses, hence the need for faster probe scheduling. Unlike our *Upper* technique [Bruno et al. 2002b] (see Section 4.2), *MPro* defines a fixed schedule of accesses to *R-Sources* during an initial object-sampling step, and thus selects which object to probe next during query execution but avoids source selection on a per-object basis. For completeness, we evaluate *MPro* experimentally in Section 7.1.3. However, as we discuss in that section, the object sampling on which *MPro* relies is problematic for a web scenario, since *SR-Sources* on the web do not typically support random sampling. In the same paper, Chang and Hwang briefly discussed parallelization techniques for *MPro* and proposed the *Probe-Parallel-MPro* algorithm, which sends one probe per object for the *k* objects with the highest score upper bounds. We adapt this algorithm to our settings and evaluate it experimentally in Section 7.2.2. A second proposed parallelization of *MPro*, *Data-Parallel MPro*, partitions the objects into several processors and merges the results of each processor's individual top-*k* computations. This parallelization is not applicable to our scenario where remote autonomous web sources "handle" specific attributes of *all* objects.

Over relational databases, Carey and Kossmann [1997, 1998] presented techniques to optimize top-*k* queries when the scoring is done through a traditional SQL order-by clause. Donjerkovic and Ramakrishnan [1999] proposed a probabilistic approach to top-*k* query optimization. Bruno et al. [2002a] 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 [2002] used a sampling-based approach to translate top-*k* queries over relational data into approximate range queries.

Additional related work includes the PREFER system [Hristidis et al. 2001], which uses pre-materialized views to efficiently answer ranked preference queries over commercial DBMSs. Natsev et al. [2001] proposed incremental algorithms to compute top-*k* queries with user-defined join predicates over sorted-access sources. The WSQ/DSQ project [Goldman and Widom 2000] 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. This *asynchronous iteration* is closely related to our handling of concurrent accesses to sources in Section 5. Finally, Avnur and Hellerstein [2000] introduced "Eddies", a query processing mechanism that reorders operator evaluation in query plans. This work shares the same design philosophy as *Upper* and *pUpper* (Sections 4.2 and 5.3), where we dynamically choose the sources to access next for each object depending on previously extracted probe information (and other factors).

4. SEQUENTIAL TOP-*K* QUERY PROCESSING STRATEGIES

In this section, we present *sequential* strategies for evaluating top-*k* queries, as defined in Section 2. In a sequential setting, a strategy can have at most one outstanding (random- or sorted-access) probe at any given time. When a probe completes, a sequential strategy chooses either to perform sorted access on a source to potentially obtain unseen objects, or to pick an already seen object, together with a source for which the object has not been probed, and perform a random-access probe on the source to get the corresponding score for the object. We discuss an existing strategy in Section 4.1 and present a sequential top-*k* query processing technique in Section 4.2. Our new technique will then serve as the basis for our parallel query processing algorithm of Section 5.

4.1 An Existing Sequential Strategy

We now review an existing algorithm to process top-*k* queries over sources that provide sorted and random access interfaces. Specifically, in Section 4.1.1, we discuss Fagin et al.'s *TA* algorithm [Fagin et al. 2001] and propose improvements over this algorithm in Section 4.1.2.

```
Algorithm $TA_z$ (Input: top-$k$ query $q$)
(01) Initialize $U_{unseen} = 1$. ($U_{unseen}$ is an upper bound on the score of any object not yet
        retrieved.)
(02) Repeat
(03)     For each SR-Source $D_i$ ($1 \leq i \leq n_{sr}$):
(04)         Get the best unretrieved object $t$ for attribute $A_i$ from $D_i$: $t \leftarrow$ getNext($D_i, q$).
(05)         Update $U_{unseen} = ScoreComb(s_\ell(1), \ldots, s_\ell(n_{sr}), \underbrace{1, \ldots, 1}_{n_r \ times})$,
             where $s_\ell(j)$ is the last score seen under sorted access in $D_j$. (Initially, $s_\ell(j) = 1$.)
(06)         For each source $D_j$ ($1 \leq j \leq n$):
(07)             If $t$'s score for attribute $A_j$ is unknown:
(08)                 Retrieve $t$'s score for attribute $A_j$, $s_j$, via a random probe to $D_j$:
                     $s_j \leftarrow$ getScore($D_j, q, t$).
(09)         Calculate $t$'s final score for $q$.
(10)         If $t$'s score is one of the top-$k$ scores seen so far, keep object $t$ along with its score.
(11) Until we have seen at least $k$ objects and $U_{unseen}$ is no larger than the scores of the
        current $k$ top objects.
(12) Return the top-$k$ objects along with their score.
```

Fig. 1. Algorithm TA_z.

4.1.1 *The TA Strategy.* Fagin et al. [2001] presented the *TA* algorithm for processing top-*k* queries over *SR-Sources*. We adapted this algorithm in Bruno et al. [2002b] and introduced the *TA-Adapt* algorithm, which handles one *SR-Source* and any number of *R-Sources*. Fagin et al. [2003] generalized *TA-Adapt* to handle any number of *SR-Sources* and *R-Sources*. Their resulting algorithm, TA_z, is summarized in Figure 1.

At any point in time, TA_z keeps track of U_{unseen}, the highest possible score an object that has not yet been seen by the algorithm can have. TA_z proceeds in the following way: for each *SR-Source*, the algorithm retrieves the next "best" object via sorted access (Step (4)), probes all unknown attribute scores for this object via random access (Steps (6)–(8)) and computes the object's final score (Step (9)). At any given point in time, TA_z keeps track of the k known objects with the highest scores. As soon as no unretrieved object can have a score higher that the current top-*k* objects, the solution is reached (Step (11)) and the top-*k* objects are returned (Step (12)). The original version of TA_z assumes bounded buffers [Fagin et al. 2003] to minimize space requirements and discards information on objects whose final scores are too low to be top-*k*. This may lead to redundant random accesses when such objects are retrieved again from a different *SR-Source*. To avoid redundant accesses, a simple solution—which we use in our implementation—is to keep all object information until the algorithm returns, which requires space that is linear in the number of objects retrieved.

4.1.2 *Improvements over TA.* Fagin et al. [2003] showed that *TA* and TA_z are "instance optimal" with respect to the family of top-*k* query processing algorithms that do not make wild guesses (see Section 4.3.2). Specifically, the *TA* and TA_z execution times are *within a constant factor* of the execution times of any such top-*k* algorithm. However, it is possible to improve on *TA* and TA_z by

```
Algorithm TA_z-EP (Input: top-k query q)
(01) Initialize $U_{unseen} = 1$. ($U_{unseen}$ is an upper bound on the score of any object not yet
     retrieved.)
(02) Repeat
(03)     For each SR-Source $D_i$ ($1 \leq i \leq n_{sr}$):
(04)         Get the best unretrieved object $t$ for attribute $A_i$ from $D_i$: $t \leftarrow \texttt{getNext}(D_i, q)$.
(05)         Update $U_{unseen} = ScoreComb(s_\ell(1), \ldots, s_\ell(n_{sr}), \underbrace{1, \ldots, 1}_{n_r\ times})$,
             where $s_\ell(j)$ is the last score seen under sorted access in $D_j$. (Initially, $s_\ell(j) = 1$.)
(06)         For each source $D_j$ ($1 \leq j \leq n$) in decreasing order of $Rank(D_j)$:
(07)             If $U(t)$ is less than or equal to the score of $k$ objects, skip to (11).
(08)             If $t$'s score for attribute $A_j$ is unknown:
(09)                 Retrieve $t$'s score for attribute $A_j$, $s_j$, via a random probe to $D_j$:
                     $s_j \leftarrow \texttt{getScore}(D_j, q, t)$.
(10)         Calculate $t$'s final score for $q$.
(11)         If we probed $t$ completely and $t$'s score is one of the top-$k$ scores, keep object $t$ along
             with its score.
(12) Until we have seen at least $k$ objects and $U_{unseen}$ is no larger than the scores of the current
     $k$ top objects.
(13) Return the top-$k$ objects along with their score.
```

Fig. 2. Algorithm *TA$_z$-EP*.

saving object probes. In Bruno et al. [2002b], we presented two optimizations over *TA* that can be applied over *TA$_z$*. The first optimization (*TA-Opt* in Bruno et al. [2002b]) saves random access probes when an object is guaranteed not to be part of the top-k answer (i.e., when its score upper bound is lower than the scores of the current top-k objects). This optimization is done by adding a shortcut test condition after Step (6) of *TA$_z$*. The second optimization (*TA-EP* in Bruno et al. [2002b]) exploits results on expensive-predicate query optimization [Hellerstein and Stonebraker 1993; Kemper et al. 1994]. Research in this area has studied how to process selection queries of the form $p_1 \wedge \cdots \wedge p_n$, 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 *Rank* of each predicate p_i, defined as $Rank(p_i) = \frac{1-selectivity(p_i)}{cost\text{-}per\text{-}object(p_i)}$, where $selectivity(p_i)$ is the fraction of the objects that are estimated to satisfy p_i, and $cost\text{-}per\text{-}object(p_i)$ is the average time to evaluate p_i over an object. We can adapt this idea to our framework as follows.

Let $w_1, \ldots, w_n$ be the weights of the sources $D_1, \ldots, D_n$ in the scoring function *ScoreComb*. If $e(A_i)$ is the expected score of a randomly picked object for source R_i, the expected decrease of $U(t)$ after probing source R_i for object t is $\delta_i = w_i \cdot (1 - e(A_i))$. We sort the sources in decreasing order of their *Rank*, where *Rank* for a source D_i is defined as $Rank(R_i) = \frac{\delta_i}{tR(R_i)}$. Thus, we favor fast sources that might have a large impact on the final score of an object; these sources are likely to substantially change the value of $U(t)$ fast.

We combine these two optimizations to define the *TA$_z$-EP* algorithm (Figure 2). The first optimization appears in Steps (7) and (11). The second optimization appears in Step (6).

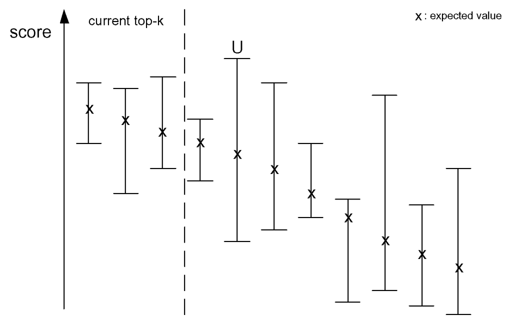


Fig. 3. Snapshot of the execution of the *Upper* strategy.

4.2 The *Upper* Strategy

We now present a top-*k* query processing strategy that we call *Upper*, a variant of which we introduced in Bruno et al. [2002b]. Our original formulation of *Upper* was for a restricted scenario of only one *SR-Source* and any number of *R-Sources*. In this article, we relax this restriction to allow for any number of *SR-Sources* and *R-Sources*. Unlike TA_z, which completely probes each object immediately after the object is identified, *Upper* allows for more flexible probe schedules in which sorted and random accesses can be interleaved even when some objects have only been partially probed. When a probe completes, *Upper* decides whether to perform a sorted-access probe on a source to get new objects, or to perform the "most promising" random-access probe on the "most promising" object that has already been retrieved via sorted access. More specifically, *Upper* exploits the following property to make its choice of probes [Bruno et al. 2002b]:

PROPERTY 4.1. *Consider a top-k query q. Suppose that at some point in time Upper has retrieved some objects via sorted access from the SR-Sources and obtained additional attribute scores via random access for some of these objects. Consider an object $t \in$ Objects whose score upper bound $U(t)$ is strictly higher than that of every other object (i.e., $U(t) > U(t')$ $\forall t' \neq t \in$ Objects), and such that t has not been completely probed. 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 score 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 3 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 4.1, 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.

```
Algorithm Upper (Input: top-k query q)
(01) Initialize $U_{unseen} = 1$, $Candidates = \emptyset$, and $returned = 0$.
(02) While ($returned < k$)
(03)     If $Candidates \neq \emptyset$, pick $t_H \in Candidates$ such that $U(t_H) = \max_{t \in Candidates} U(t)$.
(04)     Else $t_H$ is undefined.
(05)     If $t_H$ is undefined or $U(t_H) < U_{unseen}$ (unseen objects might have larger scores than
         all candidates):
(06)         Use a round-robin policy to choose the next SR-Source $D_i$ ($1 \leq i \leq n_{sr}$) to access
             via a sorted access.
(07)         Get the best unretrieved object $t$ from $D_i$: $t \leftarrow$ getNext($D_i, q$).
(08)         Update $U_{unseen} = ScoreComb(s_\ell(1), \ldots, s_\ell(n_{sr}), \underbrace{1, \ldots, 1}_{n_r \ times})$,
             where $s_\ell(j)$ is the last score seen under sorted access in $D_j$. (Initially, $s_\ell(j) = 1$.)
(09)     Else If $t_H$ is completely probed ($t_H$ is one of the top-k objects):
(10)         Return $t_H$ with its score; remove $t_H$ from Candidates.
(11)         $returned = returned + 1$.
(12)     Else:
(13)         $D_i \leftarrow SelectBestSource(t_H, Candidates)$.
(14)         Retrieve $t_H$'s score for attribute $A_i$, $s_i$, via a random probe to $D_i$:
             $s_i \leftarrow$ getScore($D_i, q, t$).
```

Fig. 4. Algorithm *Upper*.

The *Upper* algorithm is detailed in Figure 4. Exploiting Property 4.1, *Upper* chooses to probe the object with the highest score upper bound, since this object will have to be probed at least once before a top-k solution can be reached. If the score upper bound of unretrieved objects is higher than the highest score upper bound of the retrieved objects, *Upper* chooses to retrieve a new object via sorted access. In this case, *Upper* has to choose which *SR-Source* to access. This can be decided in several ways. A simple approach that works well in practice is to use a round-robin algorithm (Step (6)).

After *Upper* picks an object to probe, the choice of source to probe for the object (Step (13)) is handled by the *SelectBestSource* function, and is influenced by a number of factors: the cost of the random access probes, the weights of the corresponding attributes in the scoring function (or the ranking function itself if we consider a scoring function different than weighted sum), and the expected attribute scores.

The *SelectBestSource* function chooses the best source with which to probe object t_H next. (Object t_H is picked in Step (3).) This choice should depend on whether t_H is one of the top-k objects or not. To define this function, we would then need to know the kth highest actual score $score_k$ among all objects in *Objects*. Of course, *Upper* does not know the actual object scores a priori, so it relies on expected scores to make its choices and *estimates* the value $score_k$ (i.e., the kth top score) using $score'_k$, the kth largest *expected* object score. (We define $score'_k = 0$ if we have retrieved fewer than k objects.) We considered several implementations of the *SelectBestSource* function [Gravano et al. 2002] such as a greedy approach, or considering the best subset of sources for object t_H that is expected to decrease $U(t_H)$ below $score'_k$ (this implementation of *SelectBestSource* was presented in Bruno et al. [2002b]). Our experimental evaluation [Gravano et al. 2002] shows that using the "nonredundant sources" approach that we discuss below for *SelectBestSource* results in the best performance, so we only focus on this version of the function in this paper, for conciseness.

Our implementation of *SelectBestSource* picks the next source to probe for object t_H by first deciding whether t_H is likely to be one of the top-k objects or not:

Case 1. $E(t_H) < score'_k$. In this case, t_H is not expected to be one of the top-k objects. To decide what source to probe next for t_H, we favor sources that can have a high "impact" (i.e., that can sufficiently reduce the score of t_H so that we can discard t_H) while being efficient (i.e., with a relatively low value for tR). More specifically, $\Delta = U(t_H)$—$score'_k$ is the amount by which we need to decrease $U(t_H)$ to "prove" that t_H is not one of the top-k answers. In other words, it does not really matter how large the decrease of $U(t_H)$ is beyond Δ when choosing the best probe for t_H. Note that it is always the case that $\Delta \geq 0$: from the choice of t_H, it follows that $U(t_H) \geq score'_k$. To see why, suppose that $U(t_H) < score'_k$. Then $U(t_H) < E(t) \leq U(t)$ for k objects t, from the definition of $score'_k$. But $U(t_H)$ is highest among the objects in *Candidates*, which would imply that the k objects t such that $U(t) > U(t_H)$ had already been removed from *Candidates* and output as top-k objects. And this is not possible since the final query result has not been reached (*returned* $< k$; see Step (2)). Also, the expected decrease of $U(t_H)$ 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 attribute A_i. Then, the ratio:

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

is a good indicator of the "efficiency" of source R_i: a large value of this ratio indicates that we can reduce the value of $U(t_H)$ by a sufficiently large amount (i.e., $Min\{\Delta, \delta_i\}$) relative to the time that the associated probe requires (i.e., $tR(R_i)$).[6] Interestingly, while choosing the source with the highest rank value 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-Source*s 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 expected to be $\Delta = 0.9 - 0.5 = 0.4$. If we assume that $e(A_1) = e(A_2) = 0.5$, we would choose source R_1 (with rank $\frac{Min\{0.4, 0.05\}}{1} = 0.05$) over source R_2 (with rank $\frac{Min\{0.4, 0.45\}}{10} = 0.04$). However, we know that we will need to eventually lower $U(t)$ below $\text{score}'_k = 0.5$,

[6]*SelectBestSource* might need to be modified to handle scoring functions other than the weighted-sum function on which we focus in this article. In particular, for functions where the final object scores cannot be in general approximated or usefully bounded unless all input values are known (e.g., as is the case for the *min* function), a per-object scheduling strategy is not necessary. In such cases, the probe history of an object does not impact source choice and so, the *SelectBestSource* function should make decisions at a higher level of granularity (e.g., by ordering sources based on source access time).

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 $score'_k = 0.5$ with only one probe, 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 "nonredundant" sources. The set of redundant sources for an object 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 for object t.) To identify the subset of nonredundant available sources for object t_H, we let $\Delta = U(t_H) - score'_k$ as above and let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be the set of sources not yet probed for t_H. If $\Delta = 0$, all sources are considered not to be redundant. Otherwise, if $\Delta > 0$ we say that source R_i is redundant for object t_H at a given step of the probing process if $\forall Y \subseteq \mathcal{R} - \{R_i\}$: 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 $U(t_H)$ to $score'_k$ or lower, 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 nonredundant sources: R_i is nonredundant if and only if $\exists Y \subseteq \mathcal{R} - \{R_i\}$: $\Delta - w_i \leq \sum_{j:R_j \in Y} w_j < \Delta$. It is not difficult to prove that, for any possible assignment of values to w_i and $\Delta > 0$, there is always at least one available nonredundant source. Therefore, after identifying the subset of nonredundant sources, our *SelectBestSource* function returns the *nonredundant source* for object t_H with the maximum rank $\frac{Min\{\Delta,\delta_i\}}{tR(R_i)}$ if $\Delta > 0$. If $\Delta = 0$, all sources have the same *Rank* value, and we pick the source with the fastest random-access time for the query.

Case 2. $E(t_H) \geq score'_k$. In this case, t_H is expected to be one of the top-k objects, and so we will need to probe t_H completely. Therefore *all* sources for which t_H has not been probed are nonredundant and *SelectBestSource* returns the not-yet-probed source with the highest $\frac{\delta_i}{tR(R_i)}$ ratio.

In summary, when a probe completes, *Upper* can either (a) perform a sorted-access probe on a source if the unseen objects have the highest score upper bound (Steps (5)–(8)), or (b) select both an object and a source to probe next (Steps (12)–(14)), guided in both cases by Property 4.1. In addition, *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 (Steps (9)–(11)).

4.3 Cost Analysis

We now discuss the efficiency of the various algorithms. Specifically, Section 4.3.1 analyzes the number of sorted accesses that each algorithm requires, and Section 4.3.2 discusses the optimality of *Upper*.

4.3.1 *Counting Sorted Accesses.* Interestingly, *Upper* and *TA*$_z$ behave in an identical manner with respect to sorted accesses:

LEMMA 4.2. *Consider a top-k query q over multiple SR-Sources and R-Sources. Then, Upper and all variations of TA_z perform the same number of sorted accesses when processing q.*

PROOF. We note that the choice of sorted-access sources in both TA_z and *Upper* follows the same fixed round-robin strategy, which is independent of the input (see Step (3) for TA_z in Figure 1 and Step (6) for *Upper* in Figure 4). Therefore, after *Upper* or TA_z perform some equal number of sorted accesses, the value of U_{unseen} is the same for both algorithms. Consider the execution of both TA_z and *Upper* after both algorithms have retrieved the same set *Retrieved* of objects, with $|Retrieved| \geq k$. (Naturally, TA_z and *Upper* need to retrieve at least k objects via sorted access to output the top-k solution.)

—If TA_z decides to retrieve a new object after processing the objects in *Retrieved*, then it holds that $U_{unseen} > Score(q, m)$, where m is the object in *Retrieved* with the kth largest score. Suppose that the execution of *Upper* finishes without retrieving any new object beyond those in *Retrieved*, and let m' be the kth object output as *Upper*'s result for q. Since m' was also retrieved by TA_z, and because of the choice of m, it holds that $Score(q, m) = Score(q, m')$. Then $Score(q, m') < U_{unseen}$ and hence *Upper* could never have output this object as part of the query result (see Step (5) in Figure 4), contradicting the choice of m'. Therefore *Upper* also needs to retrieve a new object, just as TA_z does.

—If *Upper* decides to retrieve a new object after processing the objects in *Retrieved*, then it holds that *Upper* output fewer than k objects from *Retrieved* as part of the query result, and that $U(t) < U_{unseen}$ for each object $t \in Retrieved$ not yet output (see Step (5) in Figure 4). Then, since $Score(q, t) \leq U(t)$ for each object t, it follows that $Score(q, m) < U_{unseen}$, where m is the object in *Retrieved* with the kth largest actual score for q. Therefore, from Step (11) in Figure 1, it follows that TA_z also needs to retrieve a new object, just as *Upper* does. □

Interestingly, since TA_z performs *all* random accesses for the objects considered, *Upper* never performs more random accesses than TA_z does.

4.3.2 *Instance Optimality.* As presented in Fagin et al. [2003], TA_z is "instance optimal," where the definition of "instance optimality"—slightly adapted from [Fagin et al. 2003] to match our terminology—is:

Definition 4.3 (*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 required by algorithm a over the sources in D.

An interesting observation is that the number of random accesses in TA_z is an upper bound on the number of random accesses in TA_z*-EP*: TA_z*-EP* is an optimization over TA_z aimed at reducing the number of random accesses. The shortcuts used in *TA_z-EP* are only used to discard objects sooner than in *TA_z* and do not affect the number of sorted accesses performed by the algorithm. Also, as explained in the previous section, *Upper* performs no more sorted or random accesses than *TA_z* does. Hence, the *TA_z* "instance optimality" also applies to the *TA_z-EP* and *Upper* algorithms. Therefore, the experimental section of the paper (Section 7), in which we compare the *TA_z* and *Upper* algorithms, will evaluate the algorithms with real-world and synthetic data to measure their "absolute" efficiency (they are all "instance optimal").

5. PARALLEL TOP-*k* QUERY PROCESSING STRATEGIES

So far, we have discussed *sequential* top-*k* query processing strategies. As argued earlier in the article, these strategies are bound to require unnecessarily long query processing times, since web accesses usually exhibit high and variable latency. Fortunately, web sources can be probed in parallel, and also each source can typically process concurrent requests. In this section, we use the sequential strategies of Section 4 as the basis to define *parallel* query processing algorithms. We start in Section 5.1 by extending our source model to capture constraints that the sources might impose on the number of parallel requests that can be outstanding at any point in time. Then, we introduce query processing algorithms that attempt to maximize source-access parallelism to minimize query response time, while observing source-access constraints. Specifically, in Section 5.2, we discuss a simple adaptation of the *TA_z* algorithm to our parallel setting with source-access constraints. Then, in Section 5.3, we present an algorithm based on *Upper* that considers source congestion when making its probing choices. As we will see, this algorithm is robust and has the best performance in our experimental evaluation of the techniques in Sections 6 and 7.

5.1 Source Access Constraints

On the web, sources can typically handle multiple queries in parallel. However, query processing techniques must avoid sending large numbers of probes to sources. More specifically, our query processing strategies must be aware of any access restrictions that the sources in a realistic web environment might impose. Such restrictions might be due to network and processing limitations of a source, which might bound the number of concurrent queries that it can handle. This bound might change dynamically, and could be relaxed (e.g., at night) when source load is lower.

Definition 5.1 (*Source Access Constraints*). Let R be a source that supports random accesses. We refer to the **maximum** number of concurrent random accesses that a top-*k* query processing technique can issue to R as ***pR*(*R*)**, where $pR(R) \geq 1$. In contrast, sorted accesses to a source are sequential by nature (e.g., matches 11–20 are requested only after matches 1–10 have been computed and returned), so we assume that we submit `getNext` requests to a source sequentially when processing a query. However, random accesses can proceed concurrently with sorted access: we will have at most one outstanding sorted access request to a specific *SR-Source* S at any time, while we can have

```
Function SelectBestSubset (Input: object t)
(1) If we have seen $k$ or more objects through sorted access, let $t'$ be the object with the $k$th
    largest expected score, and let $score'_k = E(t')$.
(2) Else $score'_k = 0$.
(3) If $E(t) \geq score'_k$:
(4)     Define $S \subseteq \{D_1, \ldots, D_n\}$ as the set of all sources not yet probed for $t$.
(5) Else:
(6)     Define $S \subseteq \{D_1, \ldots, D_n\}$ as the set of sources not yet probed for $t$ such that
        (i) $U(t) < score'_k$ if each source $D_j \in S$ were to return the expected value for $t$, and
        (ii) the time $\sum_{D_j \in S} eR(D_j, t)$ is minimum among the source sets with this
        property (see text).
(7) Return $S$.
```

Fig. 5. Function *SelectBestSubset.*

up to $pR(S)$ outstanding random-access requests to this same source, for a total of up to $1 + pR(S)$ concurrent accesses.

Each source D_i can process up to $pR(D_i)$ random accesses concurrently. Whenever the number of outstanding probes to a source D_i falls below $pR(D_i)$, a parallel processing strategy can decide to send one more probe to D_i.

5.2 Adapting the *TA* Strategy

The *TA* algorithm (Section 4.1.1) as described by Fagin et al. [2001] does not preclude parallel executions. We adapt the TA_z version of this algorithm [Fagin et al. 2003] to our parallel scenario and define *pTA*, which probes objects in parallel in the order in which they are retrieved from the *SR-Sources*, while respecting source-access constraints. Specifically, each object retrieved via sorted access is placed in a queue of discovered objects. When a source D_i becomes available, *pTA* chooses which object to probe next for that source by selecting the first object in the queue that has not yet been probed for D_i. Additionally, *pTA* can include the *TA-Opt* optimization over TA_z to stop probing objects whose score cannot exceed that of the best top-*k* objects already seen (Section 4.1.2). *pTA* then takes advantage of all available parallel source accesses to return the top-*k* query answer as fast as possible. However, it does not make choices on which probes to perform, but rather only saves probes on "discarded" objects.

5.3 The *pUpper* Strategy

A parallel query processing strategy might react to a source D_i having fewer than $pR(D_i)$ outstanding probes by picking an object to probe on D_i. A direct way to parallelize the *Upper* algorithm suggests itself: every time a source D_i becomes underutilized, we pick the object t with the highest score upper bound among those objects that need to be probed on D_i in accordance with (a variation of) *Upper*. We refer to the resulting strategy as *pUpper*.

To select which object to probe next for a source D_i, *pUpper* uses the *SelectBestSubset* function shown in Figure 5, which is closely related to the *SelectBestSource* function of the sequential *Upper* algorithm of Section 4.2. The sequential *Upper* algorithm uses the *SelectBestSource* function to pick the single best source for a given object. Only one source is chosen each time because the algorithm is sequential and does not allow for multiple concurrent probes to proceed simultaneously. In contrast, probes can proceed concurrently in a parallel setting and this is reflected in the *SelectBestSubset* function, which generalizes *SelectBestSource* and picks a minimal set of sources that need to be probed for a given object. Intuitively, these *multiple* probes might proceed in parallel to speed up query execution. When a random-access source D_i becomes underutilized, we identify the object t with the highest score upper bound such that $D_i \in \mathit{SelectBestSubset}(t)$.

The *SelectBestSubset* function attempts to predict what probes will be performed on an object t before the top-k answer is reached: (1) if t is expected to be one of the top-k objects, all random accesses on sources for which t's attribute score is missing will be considered (Step (4)); otherwise (2) only the fastest subset of probes expected to help discard t—by decreasing t's score upper bound below the kth highest (expected) object score $score'_k$—are considered (Step (6)). *SelectBestSubset* bases its choices on the known attribute scores of object t at the time of the function invocation, as well as on the *expected access time* $eR(D_j, t)$ for each source D_j not yet probed for t, which is defined as the sum of two terms:

(1) The time $wR(D_j, t)$ that object t will have to "wait in line" before being probed for D_j: any object t' with $U(t') > U(t)$ that needs to be probed for D_j will do so before t. Then, if $\mathit{precede}(D_j, t)$ denotes the number of such objects, we can define $wR(D_j, t) = \lfloor \frac{\mathit{precede}(D_j,t)}{pR(D_j)} \rfloor \cdot tR(D_j)$. To account for the waiting time wR and the $\mathit{precede}(D_j, t)$ value for all sources accurately, objects are considered in decreasing order of their score upper bounds.
(2) The time $tR(D_j)$ to actually perform the probe.

The time $eR(D_j, t)$ is then equal to:

$$\begin{aligned} eR(D_j, t) &= wR(D_j, t) + tR(D_j) \\ &= tR(D_j) \cdot \left(\left\lfloor \frac{\mathit{precede}(D_j, t)}{pR(D_j)} \right\rfloor + 1 \right) \end{aligned}$$

Without factoring in the wR waiting time, all best subsets tend to be similar and include only sources with high weight in the query and/or with low access time tR. Considering the waiting time is critical to *dynamically account for source congestion*, and allows for slow sources or sources with low associated query weight to be used for some objects, thus avoiding wasting resources by not taking advantage of all available concurrent source accesses. The fastest subset of probes expected to help discard t is chosen based on the *sum* of the expected access time of its associated sources. While using their *maximum* value would give a better estimation of the expected time to probe all sources in the subset, the *sum* function helps to take into consideration the global source congestion that would result from probing the subset.

As mentioned before, this *SelectBestSubset* function is closely related to the *SelectBestSource* function of Section 4.2. Both functions allow for dynamic query evaluation by relying on current available information on object scores

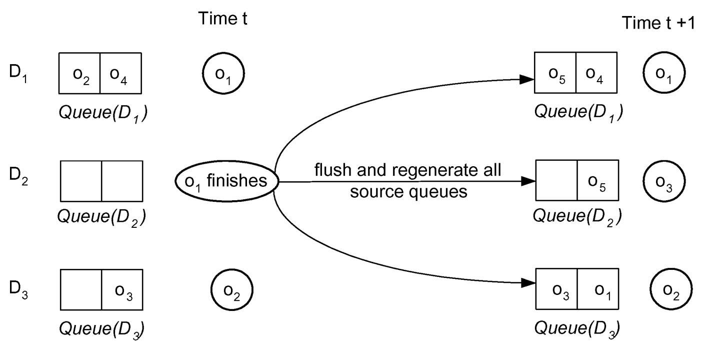


Fig. 6. An execution step of *pUpper*.

to make probing choices. However, *SelectBestSubset* is used in a parallel setting where probes can be issued concurrently, so there is no need to determine a total order of the source probes for each object and "subset" probes can be issued concurrently. Therefore, the *Rank* metric presented in Section 4.2 is not strictly needed in the *SelectBestSubset* function. Interestingly, in the specific scenario where any one source is expected to be enough to discard an object t, *SelectBestSubset* selects the same source for t as *SelectBestSource* would if we ignore the source waiting time: in this scenario, any source is expected to decrease the score upper bound of t by at least Δ (Section 4.2), and *SelectBest-Subset* picks the fastest such source. This choice is equivalent to selecting the source with the highest $\frac{Min\{\Delta,\delta_i\}}{tR(R_i)}$ rank value, as is done by *SelectBestSource*.

The query-processing strategy above is expensive in local computation time: it might require several calls to *SelectBestSubset* each time a random-access source becomes available, and *SelectBestSubset* takes time that is exponential in the number of sources. To reduce local processing time, we devise an efficient algorithm based on the following observation: whenever *SelectBestSubset* is invoked to schedule probes for a source D_i, information on the best probes to perform for D_i *as well as for other sources* is computed. Scheduling probes for just one source at any given time results in discarding the information on valuable probes to the other sources, which results in redundant computation when these other sources become underutilized and can then receive further probes.

With the above observations in mind, our parallel top-*k* processing algorithm, *pUpper*, precomputes sets of objects to probe for each source. When a source becomes available, *pUpper* checks whether an object to probe for that source has already been chosen. If not, *pUpper* recomputes objects to probe for *all sources*, as shown in Figure 6. This way, earlier choices of probes on any source might be revised in light of new information on object scores: objects that appeared "promising" earlier (and hence that might have been scheduled for further probing) might now be judged less promising than other objects after some probes complete. By choosing several objects to probe for every source in a single computation, *pUpper* drastically reduces local processing time.

The *pUpper* algorithm (Figure 7) associates a queue with each source for random access scheduling. The queues are regularly updated by calls to the function *GenerateQueues* (Figure 8). During top-*k* query processing, if a source

```
Algorithm pUpper (Input: top-$k$ query $q$)
(01) Repeat
(02)     For each SR-Source $D_i$ $(1 \leq i \leq n_{sr})$:
(03)         If no sorted access is being performed on $D_i$ and more objects are available from $D_i$
             for $q$ :
(04)             Call pGetNext$(D_i, q)$ asynchronously.
(05)     For each source $D_i$ $(1 \leq i \leq n)$:
(06)         While fewer that $pR(D_i)$ random accesses are being performed on $D_i$:
(07)             If $Queue(D_i) = \emptyset$:
(08)                 $GenerateQueues()$.
(09)             Else:
(10)                 $t = Dequeue(D_i)$.
(11)                 Call pGetScore$(D_i, q, t)$ asynchronously.
(12) Until we have identified $k$ top objects
(13) Return the top-$k$ objects along with their scores.
```

Fig. 7. Algorithm *pUpper*.

```
Function Generate Queues ()
(1) Let Considered be the set of "alive" objects (i.e., objects whose score upper bound is greater
    than the $k$th largest score lower bound).
(2) For each source $D_i$ $(1 \leq i \leq n)$:
(3)     Empty $Queue(D_i)$.
(4) While $Considered \neq \emptyset$ and $\exists i \in \{1, ..., n\} : |Queue(D_i)| < L$:
(5)     Extract $t_H$ from Considered such that: $U(t_H) = \max_{t \in Considered} U(t)$.
(6)     $S = SelectBestSubset(t_H)$.
(7)     For each source $D_j \in S$:
(8)         If $|Queue(D_j)| < L$: $Enqueue(D_j, t_H)$.
```

Fig. 8. Function *GenerateQueues*.

D_i is available, *pUpper* checks the associated random-access queue *Queue*(D_i). If *Queue*(D_i) is empty, then all random access queues are regenerated (Steps (7)–(8) in Figure 7). If *Queue*(D_i) is not empty, then simply a probe to D_i on the first object in *Queue*(D_i) is sent (Steps (9)–(11)). To avoid repeated calls to *GenerateQueues* when a random access queue is continuously empty (which can happen, for example, if all known objects have already been probed for its associated source), a queue left empty from a previous execution does not trigger a new call to *GenerateQueues*.

As sorted accesses are sequential in nature (Definition 5.1, Section 5.1), *pUpper* attempts to always have exactly one outstanding sorted-access request per *SR-Source* D_i (Steps (2)–(4)). As soon as a sorted access to D_i completes, a new one is sent until all needed objects are retrieved from D_i.

Source accesses are performed by calling `pGetNext` and `pGetScore`, which are asynchronous versions of the `getNext` and `getScore` source interfaces (Definition 2.1); these asynchronous calls, similar to the *asynchronous iteration* described in WSQ/DSQ [Goldman and Widom 2000], allow the query processing algorithm to continue without waiting for the source accesses to complete. `pGetNext` and `pGetScore` send the corresponding probes to the sources, wait for their results to return, and update the appropriate data structures with the new information. Of course, *pUpper* keeps track of outstanding probes so as not to issue duplicate probes. The top-*k* query processing terminates when the top-k objects are identified, which happens when no object can have a final score greater than that of any of the current top-k objects.

To allow for dynamic queue updates at regular intervals, and to ensure that queues are generated using recent information, we define a parameter L that indicates the length of the random-access queues generated by the *GenerateQueues* function. A call to *GenerateQueues* to populate a source's random-access queue provides up-to-date information on current best objects to probe for all sources, therefore *GenerateQueues* regenerates all random-access queues. An object t is only inserted into the queues of the sources returned by the *SelectBestSubset(t)* function from Figure 5 (Steps (6)–(8) in Figure 8). Additionally, objects are considered in the order of their score upper bound (Step (5)), considering only "alive" objects, that is, objects that have not been discarded (Step (1)).

pUpper precomputes a list of objects to access per source, based on expected score values. Of course, the best subset for an object might vary during processing, and *pUpper* might perform "useless" probes. Parameter L regulates the tradeoff between queue "freshness" and local processing time, since L determines how frequently the random access queues are updated and how reactive *pUpper* is to new information.

6. EVALUATION SETTING

In this section, we discuss data structures that we use to implement the query processing strategies of Sections 4 and 5 (Section 6.1). We also define the synthetic and real data sets (Section 6.2) that we use for the experimental evaluation of the various techniques, as well as the prototype that we implemented for our experiments over real web-accessible sources (Section 6.3). Finally, we discuss the metrics and other settings that we use in our experimental evaluation (Section 6.4).

6.1 Supporting Data Structures

Our algorithms keep track of the objects retrieved and their partial score information in a hash table indexed by the object IDs. For each object, we record the attribute scores returned by the different sources (a special value is used when information about a particular source is not yet available). For efficiency, we also incrementally maintain the score upper bounds of each object. Finally, depending on the algorithm, each object is augmented with a small number of pointers that help us to efficiently maintain the rank of each object in different ordered lists (see Gravano et al. [2002]). During the execution of the algorithms of Sections 4 and 5, each object can be part of multiple sorted lists. As an example, *Upper* (Section 4.2) needs to keep track of the object with the largest score upper bound (Step (3) in the algorithm in Figure 4). The *SelectBestSource* and *SelectBestSubset* functions also need to identify the object with the kth 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. We additionally modified these standard priority queues to extract in constant time the kth ranked object in the list still with logarithmic-time insertions and deletions (we refer the reader to Gravano et al. [2002] for further implementation details).

6.2 Local Sources

We generate a number of synthetic *SR-Sources* and *R-Sources* for our experiments. The attribute values for each object are generated using one of the following distributions:

Uniform. Attributes are independent of each other and attribute values are uniformly distributed (default setting).

Gaussian. Attributes are independent of each other and attribute values are generated from five overlapping multidimensional Gaussian bells [Williams et al. 1993].

Zipfian. Attributes are independent of each other and attribute values are generated from a Zipf function with 1,000 distinct values and Zipfian parameter $z = 1$. The 1,000 distinct attribute values are generated randomly in the [0,1] range, and the ith most frequent attribute value appears $f(i) = |Objects|/(i^z \cdot \sum_{j=1}^{1,000} 1/j^z)$ times.

Correlated. We divide sources into two groups and generate attribute values so that values from sources within the same group are correlated. In each group, the attribute values for a "base" source are generated using a uniform distribution. The attribute values for the other sources in a group are picked for an object from a short interval around the object's attribute value in the "base" source. Our default *Correlated* data set consists of two groups of three sources each.

Mixed. Attributes are independent of each other. Sources are divided into three groups, and the attribute values within each group are generated using the *Uniform*, *Gaussian*, and *Zipfian* distributions, respectively.

Cover. To validate our techniques on real data distributions, we performed experiments over the *Cover* data set, a six-dimensional projection of the CovType data set [Blake and Merz 1998], used for predicting forest cover types from cartographic variables. The data contains information about various wilderness areas. Specifically, we consider six attributes: elevation (in meters), aspect (in degrees azimuth), slope (in degrees), horizontal distance to hydrology (in meters), vertical distance to hydrology (in meters), and horizontal distance to roadways (in meters). We extracted a database of 10,000 objects from the CovType data set.

For simplicity, we will refer to the synthetic and the *Cover* sources as the "local" sources, to indicate that these are locally available sources under our control, as opposed to the real web sources described next. For our experiments, we vary the number of *SR-Sources* n_{sr}, the number of *R-Sources* n_r, the number of objects available through sorted access $|Objects|$, the random access time $tR(D_i)$ for each source D_i (a random value between 1 and 10), the sorted access time $tS(D_i)$ for each source D_i (a random value between 0.1 and 1), and the

Table I. Default Parameter Values for Experiments Over Synthetic Data

k	n_{sr}	n_r	$\|Objects\|$	tR	tS	pR	*Data Sets*
50	3	3	10,000	[1, 10]	[0.1, 1]	5	Uniform

Table II. Real Web-Accessible Sources Used in the Experimental Evaluation

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

maximum number of parallel random accesses $pR(D_i)$ for each source D_i (for the parallel algorithms only). Table I lists the default value for each parameter. Unless we specify otherwise, we use this default setting.

6.3 Real Web-Accessible Sources

In addition to experiments over the "local" data sets above, we evaluated the algorithms over real, autonomous web sources. For this, we implemented a prototype of the algorithms to answer top-*k* queries about New York City restaurants. Our prototype is written in C++ and Python, using C++ threads and multiple Python subinterpreters to support concurrency for parallel algorithms.

6.3.1 *Attributes.* Users input a starting address and their desired type of cuisine (if any), together with importance weights for the following *R-Source* attributes: *SubwayTime* (handled by the SubwayNavigator site[7]), *DrivingTime* (handled by the MapQuest site), *Popularity* (handled by the AltaVista search engine[8]; see below), *ZFood*, *ZService*, *ZDecor*, and *ZPrice* (handled by the Zagat-Review web site), and *TRating* and *TPrice* (provided by the New York Times's NYT-Review web site). The Verizon Yellow Pages listing,[9] which for sorted access returns restaurants of the user-specified type sorted by shortest distance from a given address, is the only *SR-Source*. Table II summarizes these sources and their interfaces.

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 the *ZPrice* and *TPrice* target values are set to the least expensive value in the scale. In the default setting, the weights of all six sources are equal. The *Popularity* attribute requires further explanation. We approximate the "popularity" of a restaurant with the number of web pages that mention the

[7]`http://www.subwaynavigator.com.`
[8]`http://www.altavista.com.`
[9]`http://www.superpages.com.`

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 [Goldman and Widom 2000].) Consider, for example, restaurant "Tavern on the Green," which is one of the most popular restaurants in the United States. As of the writing of this article, a query on AltaVista on "Tavern on the Green" AND "New York" returns 4,590 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 24 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.

6.3.2 *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.

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 good performance, Mills [1983] 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 sorted accesses to the *SR-Sources* S_i are decided independently of their sorted-access times, we do not adjust $tS(S_i)$.

6.4 Evaluation Metrics and Other Experimental Settings

To understand the relative performance of the various top-k processing techniques over local sources, we time the two main components of the algorithms:

—t_{probes} is the time spent accessing the remote sources, in "units" of time. (In Section 7.1.2, we report results for different values—in ms—of this time unit.)

—t_{local} is the time spent locally scheduling remote source accesses, in seconds.

While source access and local scheduling happen in parallel, it is revealing to analyze the t_{probes} and t_{local} times associated with the query processing techniques separately, since the techniques that we consider differ significantly in the amount of local processing time that they require. For the experiments over the real-web sources, we report the total query execution time:

—t_{total} is the total time spent executing a top-k query, in seconds, including both remote source access and scheduling.

We also report the number of random probes issued by each technique[10]:

—$|probes|$ is the total number of random probes issued during a top-k query execution.

Finally, we quantify the extent to which the parallel techniques exploit the available source-access parallelism. Consider *Upper*, the sequential algorithm that performed the best for our web-source scenario (with relatively expensive probes and no information on the underlying data distribution known in advance) according to the experimental evaluation in Section 7.1. Ideally, parallel algorithms would keep sources "humming" by accessing them in parallel as much as possible. At any point in time, up to $n_{sr} + \sum_{i=1}^{n} pR(D_i)$ concurrent source accesses can be in progress. Hence, if t_{Upper} is the time that *Upper* spends accessing remote sources sequentially, then $t_{Upper}/(n_{sr} + \sum_{i=1}^{n} pR(D_i))$ is a (loose) lower bound on the parallel t_{probes} time for the parallel algorithms, assuming that parallel algorithms perform at least as many source accesses as *Upper*. To observe what fraction of this potential parallel speedup the parallel algorithms achieve, we report:

$$Parallel\ Efficiency = \frac{t_{Upper}/(n_{sr} + \sum_{i=1}^{n} pR(D_i))}{t_{probes}}$$

A parallel algorithm with *Parallel Efficiency* = 1 manages to essentially fully exploit the available source-access parallelism. Lower values of *Parallel Efficiency* indicate that either some sources are left idle and not fully utilized during query processing, or that some additional probes are being performed by the parallel algorithm.

For the local sources, unless we note otherwise we generate 100 queries randomly, with attribute weights randomly picked in the [1,10] range. We report the average values of the metrics for different settings of n_{sr}, n_r, $|Objects|$, pR, and k for different attribute distributions. We conducted experiments on 1.4-Ghz 2-Gb RAM machines running Red Hat Linux 7.1.

[10]For sequential algorithms, the number of sorted accesses is the same for all presented techniques (Section 4.3.1). For parallel algorithms, we observed comparable number of sorted accesses across techniques.

Table III. "Dimensions" to Characterize Sequential Query Processing Algorithms

		Per-Query Scheduling of Probes	Per-Object Scheduling of Probes
No Interleaving of Probes across Objects		*TA_z-EP*	*TA_z-SelectBestSource*
Interleaving of Probes across Objects	No Sampling Available	*MPro-EP*	*Upper*
	Sampling Available	*MPro*	*Upper-Sample*

For the real web sources, we defined queries that ask for top French, Italian, and Japanese restaurants in Manhattan, for users located in different addresses. For the sequential algorithms (Section 7.1.4), attribute weights are arbitrarily picked from the [1,10] range for each query. For the parallel algorithms (Section 7.2.3), all attribute weights are equal. We report the average t_{total} value for different settings of pR and k. We conducted experiments on a 550-Mhz 758-Mb RAM machine running Red Hat Linux 7.1.

7. EVALUATION RESULTS

We now present the experimental results for the sequential (Section 7.1) and the parallel (Section 7.2) techniques, using the data and general settings described in Section 6.

7.1 Sequential Algorithms

In this section, we compare the performance of *Upper* (Section 4.2) with that of *TA_z-EP* (Section 4.1.1). *Upper* is a technique in which source probes are scheduled at a fine object-level granularity, and where probes on different objects can be interleaved (see Table III). In contrast, *TA_z-EP* is a technique in which source probes are scheduled at a coarse query-level granularity, and where each object is fully processed before probes on a different object can proceed. The *MPro* algorithm of Chang and Hwang [2002] (Section 3) is an example of a technique with interleaving of probes on different objects and with query-level probe scheduling. (We evaluate *MPro* in Section 7.1.3, where we consider a scenario in which object sampling—as required by *MPro*—is possible. We also defer the discussion of the *Upper-Sample* technique until that section.) *MPro-EP* is an instantiation of the *MPro* algorithm with a different source-order criterion. Specifically, *MPro-EP* departs from the original *MPro* algorithm in that it does not rely on object sampling and orders sources by their *Rank* values as defined in Section 4.1.2. Note that *MPro-EP* can also be regarded as a modification of *Upper* for which the *SelectBestSource* function always considers each source's object-independent *Rank* value as defined in Section 4.1.2 when deciding what source to pick for a given object.

The "dimensions" outlined in Table III suggest an additional technique. This technique, denoted as *TA_z-SelectBestSource* in Table III, is similar to *TA_z-EP* in that it does not interleave probes on multiple objects. However, the schedule of probes on each object is not fixed, but rather is influenced by the returned probe scores and determined dynamically using *Upper*'s *SelectBestSource* function.

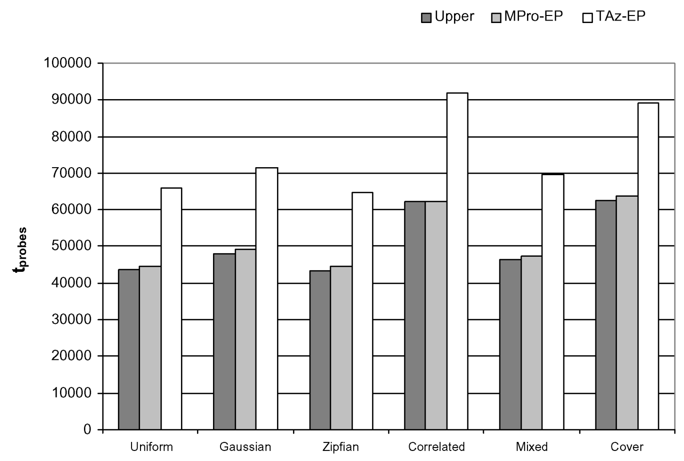


Fig. 9. Performance of the different strategies for the default setting of the experiment parameters, and for alternate attribute-value distributions.

For conciseness, we do not report experimental figures for this technique, since it results in only minor time savings over the simpler *TA$_z$-EP* algorithm. Similarly, we do not consider variations of *TA$_z$-EP* and *TA$_z$-SelectBestSource* that exploit sampling-derived information.

By comparing *MPro-EP* and *TA$_z$-EP*, our experiments help quantify the saving in probing time that is due to the interleaving of object probes. By comparing *MPro-EP* and *Upper*, our experiments help understand the impact of the relatively expensive per-object scheduling on query processing efficiency.

7.1.1 *Local Data Sets: Probing Time.* We first study the performance of the techniques when we vary the local data set parameters.

Effect of the Attribute Value Distribution. Figure 9 reports results for the default setting (Table I), for various attribute value distributions. In all cases, *Upper* substantially outperforms *TA$_z$-EP*. The performance of *MPro-EP* is just slightly worse than that of *Upper*, which suggests that the gain in probing time of *Upper* over *TA$_z$-EP* mostly results from interleaving probes on objects. Interestingly, while *Upper* has faster overall probing times that *MPro-EP*, *MPro-EP* results in slightly fewer random accesses (e.g., for the *Uniform* data set, *Upper* performed on average 11,342 random accesses and *MPro-EP* performed on average 11,045 random accesses). For the *Cover* data set, which consists of real-world data, the results are similar to those for the other data sets.

Effect of the Number of Objects Requested k. Figure 10 reports results for the default setting (Table I) as a function of *k*. 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. Once again, the *Upper* strategy consistently outperforms *TA$_z$-EP*, with *MPro-EP* as a close second.

Effect of the Number of Sources n. Figure 11 reports results for the default setting, as a function of the total number of sources *n* (half the sources are

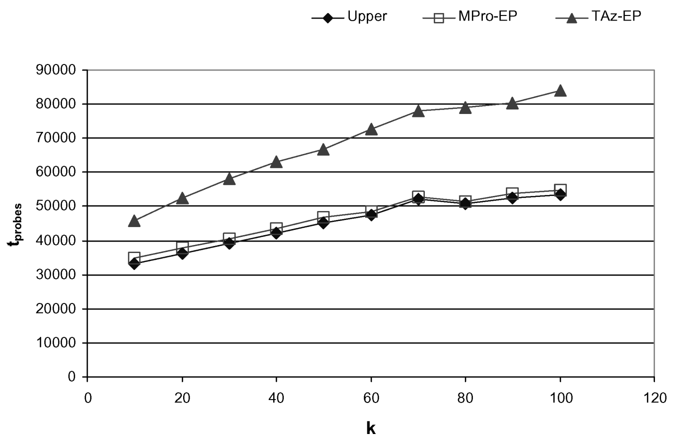


Fig. 10. Performance of the different strategies for the default setting of the experiment parameters, as a function of the number of objects requested k.

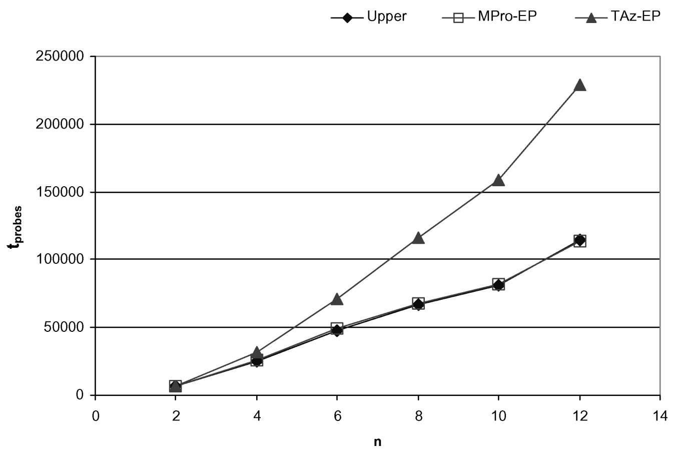


Fig. 11. Performance of the different strategies for the *Uniform* data set, as a function of the number of sources.

SR-Sources, half are *R-Sources*). Not surprisingly, the t_{probes} time needed by all the algorithms increases with the number of available sources. When we consider a single *SR-Source* and a single *R-Source*, t_{probes} is the same for all algorithms. However, when more sources are available, the differences between the techniques become more pronounced, with *Upper* and *MPro-EP* consistently resulting in the best performance.

Effect of the Number of SR-Sources n_{sr}. Figure 12 reports results for the default setting, as a function of the total number of sources n_{sr} (out of a total of six sources). The performance of *TA_z-EP* remains almost constant when we vary the number of *SR-Sources*. In contrast, the performance of *Upper* and *MPro-EP* improves when the number of *SR-Sources* is high, as more information on the top objects is obtained from sorted accesses, which are cheaper than random accesses. The information gained from these extra sorted accesses allows these

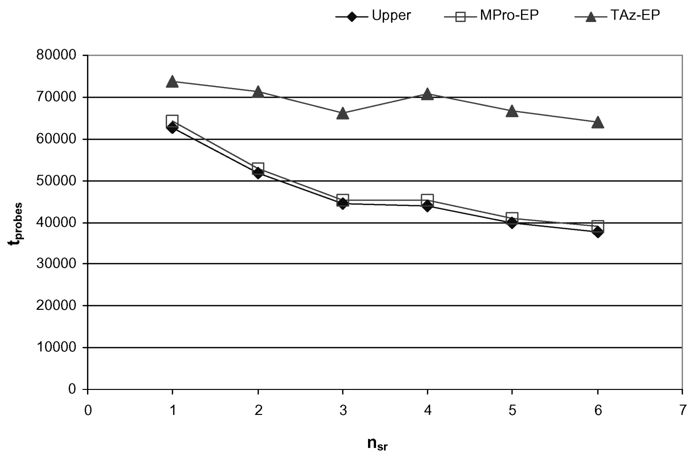


Fig. 12. Performance of the different strategies for the *Uniform* data set, as a function of the number of *SR-Sources*.

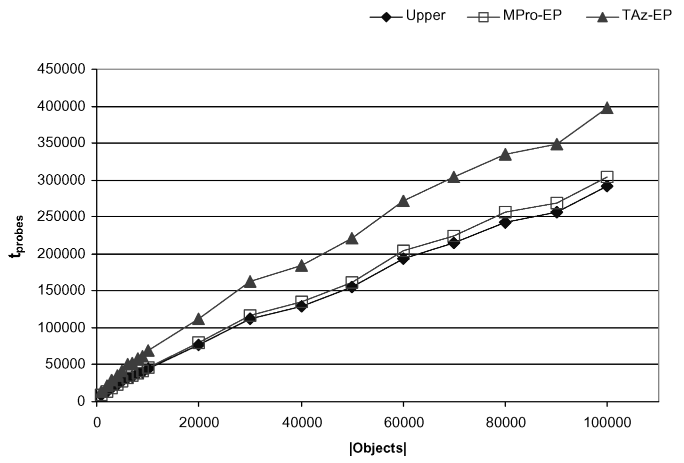


Fig. 13. Performance of the different strategies for the *Uniform* data set, as a function of the cardinality of *Objects*.

algorithms to identify high-score objects (objects with high scores for all the *SR-Sources* attributes) sooner and therefore to return the top-*k* objects faster. *Upper* is slightly better than *MPro-EP*, with savings in probing time that remains close to constant for all values of n_{sr}.

Effect of the Cardinality of the Objects Set. Figure 13 studies the impact of the number of objects available. 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 $|Objects|$. *MPro-EP* is faster and scales better than TA_z*-EP* since *MPro-EP* only considers objects that need to be probed before the top-*k* answer is reached and therefore does not waste resources on useless probes. *Upper*'s reduction in probing time over *MPro-EP* increases with the number of

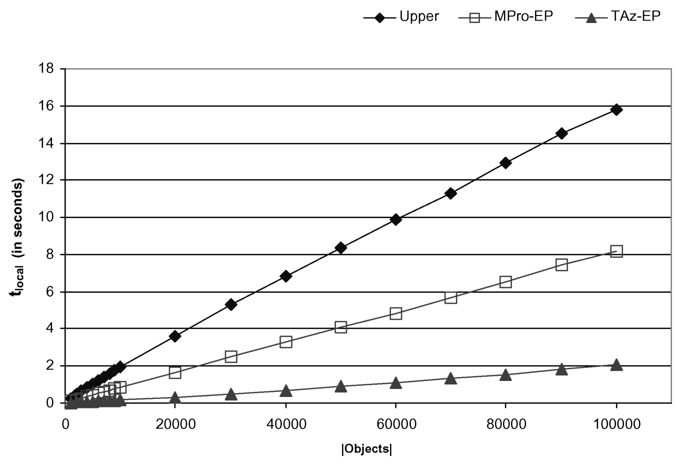


Fig. 14. The local processing time for *Upper*, *MPro-EP*, and *TA_z-EP*, as a function of the number of objects.

objects, suggesting that per-object source scheduling becomes more efficient as the number of objects increase.

7.1.2 *Local Data Sets: Local Processing Time.* In the previous section, we showed that *Upper* and *MPro-EP* result in substantially fewer random probes than *TA_z-EP*. However, probe interleaving requires expensive computation as object score information needs to be kept and sorted. In addition, *Upper* requires more expensive probe scheduling than *TA_z-EP* and *MPro-EP* do, so we now turn to studying the effect of this local computation on overall performance. Interestingly, we show experimentally that *Upper* results in considerably faster executions than *TA_z-EP*, considering both probing time and local execution time. Our experiments also show that *Upper* results in slightly faster overall query execution times than *MPro-EP*.

Figure 14 shows the t_{local} time for *Upper*, *MPro-EP*, and *TA_z-EP* for the default setting of the experiments in Table I, and for varying number of objects. Not surprisingly, *TA_z-EP* is *locally* more efficient than *Upper* and *MPro-EP*. The additional local processing needed by *Upper* and *MPro-EP* is spent maintaining object queues. (Both techniques need access to the object with the largest score upper bound at different points in time.) In turn, *Upper* is more expensive than *MPro-EP* because of two factors: (1) *Upper* schedules probes at the object level, while *MPro-EP* does so at a coarser query level, and (2) unlike *MPro-EP*, *Upper* needs fast access to the object with the kth largest expected score, for which the modified priority queue mentioned in Section 6.1 needs to be maintained. Interestingly, the second factor above accounts for most of the difference in execution time between *Upper* and *MPro-EP* according to our experiments. 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

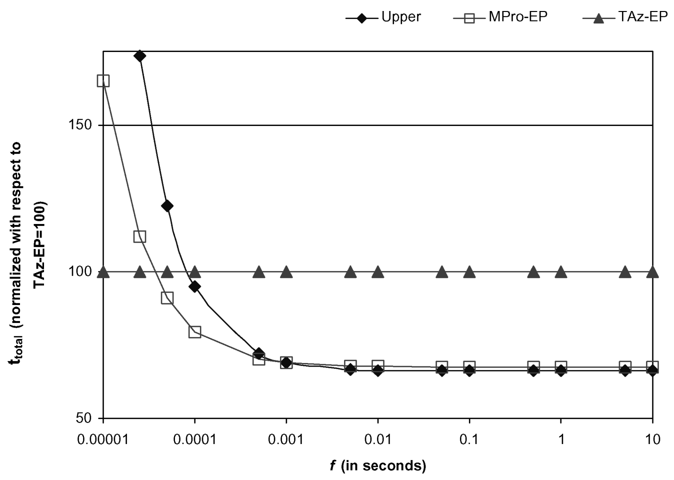


Fig. 15. The total processing time for *Upper*, *MPro-EP*, and *TA$_z$-EP*, as a function of the time unit *f*.

the random-access time *tR*. Figure 15 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$_z$-EP*. This figure shows that, for *TA$_z$-EP* to be faster than *Upper* in *total* execution time, the time unit for random accesses should be less than 0.075 ms, which translates in random access times no larger than 0.75 ms. For comparison, note that the fastest real-web random access time in our experiments was around 25 ms. For all realistic values of f, it follows that while *TA$_z$-EP* is *locally* faster than *Upper*, *Upper* is *globally* more efficient. Additionally, Figure 15 shows that *Upper* slightly outperforms *MPro-EP* for f higher than 1 ms, which means that the extra computation in the *SelectBestSource* function of *Upper* results in (moderate) savings in probing time and thus in slightly faster overall 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. In conclusion, the extra local computation required by *Upper* for selecting the best object-source pair to probe next allows for savings in total query execution time *when random-access probes are slow relative to local CPU speed*, which is likely to be the case in the web-source scenario on which we focus in this article.

7.1.3 *Local Data Sets: Using Data Distribution Statistics.* The experiments we presented so far assume that no information about the underlying data distribution is known, which forces *Upper* to rely on default values (e.g., 0.5) for the expected attribute scores (Section 4.2). We now study this aspect of *Upper* in more detail, as well as consider the scenario where additional statistics on the data distribution are available (see last row of Table III).

Effect of Average Expected Scores. In absence of reliable information on source-score distribution, our techniques initially approximate "expected" scores with the constant 0.5. (As a refinement, this value then continuously

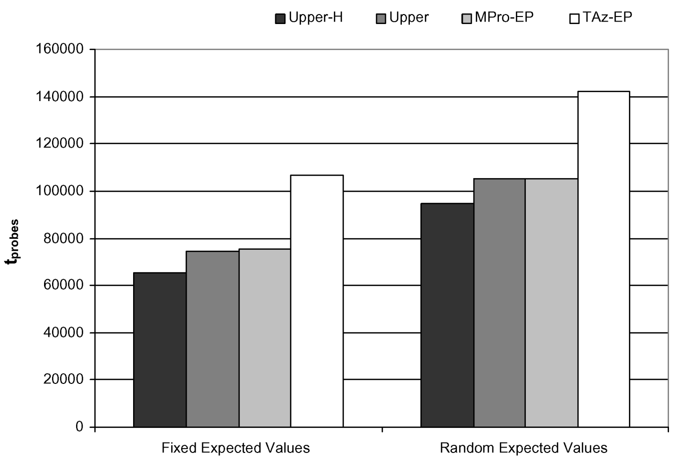


Fig. 16. The performance of *Upper* improves when the expected scores are known in advance.

decreases for *SR-Sources* as sorted accesses are performed; see Section 2.) This estimation could in principle result in bad performance when the actual average attribute scores are far from 0.5. To evaluate the effect of this choice of expected scores on the performance of *Upper*, we generate data sets with different score distributions and compare the performance of *Upper* with and without knowledge of the actual average scores. In particular, we first evaluate 100 queries using *Upper*, *MPro-EP*, and TA_z*-EP* assuming that the average scores are 0.5. Then, we evaluate the same queries, but this time we let *Upper* use the actual average scores to choose which sources to probe, progressively "shrinking" these average scores as sorted accesses are performed (see Section 2). We refer to this "hypothetical" version of *Upper* as *Upper-H*. (Note that TA_z*-EP* and *MPro-EP* do not rely on expected scores.) The results are shown in Figure 16. For the first experiment, labeled "Fixed Expected Values" in the figure, the scores for four out of the six sources are uniformly distributed between 0 and 1 (with average score 0.5), the scores for the fifth source range from 0 to 0.2 (with average score 0.1), and the scores for the sixth source range from 0.8 to 1 (with average score 0.9). For the second experiment, labeled "Random Expected Values" in the figure, the mean scores for all sources were random values between 0 and 1. Not surprisingly, *Upper-H* results in smaller t_{probes} time than *Upper*, 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_z*-EP* and comparable to that of *MPro-EP* even when *Upper* uses the default value of 0.5 as the expected attribute score.

Comparison with MPro. As discussed in Section 3, a key difference between Chang and Hwang's *MPro* algorithm [Chang and Hwang 2002] and *Upper* is that *MPro* assumes a fixed query-level schedule of sources to access as random probes, and does not base its source-order choices on the current query state. *MPro* uses sampling to determine its fixed random probe schedule [Chang and Hwang 2002]. To determine its schedule, *MPro* computes the *aggregate*

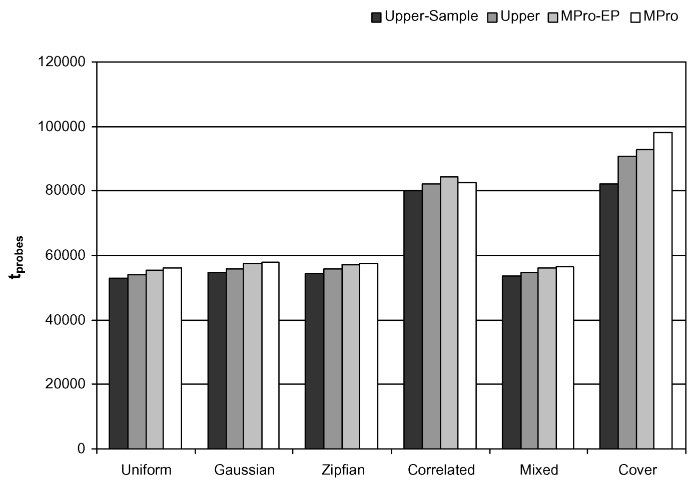


Fig. 17. Performance of *Upper-Sample*, *Upper*, *MPro-EP*, and *MPro*, when sampling is available and for different data sets.

selectivities of the various query predicates (random probes) based on the sample results.

Sampling of objects in our web scenario is problematic: *SR-Sources* on the web do not typically support random sampling, so there is no easy way to implement *MPro*'s sampling-based probe scheduling over general web sources. Still, for completeness, in this section we compare *MPro* experimentally with *Upper* and *MPro-EP* over the local data sets. Furthermore, we also evaluate a simple variation of *Upper*, *Upper-Sample*, that exploits a sample of the available objects to determine the expected score for each attribute, rather than assuming this value is 0.5. Both *MPro* and *Upper-Sample* are possible query processing techniques for scenarios in which object sampling is indeed feasible.

We experimentally compared *MPro*, *Upper*, *Upper-Sample*, and *MPro-EP*. For these experiments, we set the number of *SR-Sources* to 1. (While *MPro* could support multiple *SR-Sources* by "combining" them into a single object stream using *TA* [Chang and Hwang 2002], *MPro* would not attempt to interleave random probes on the *SR-Sources*. Hence, to make our comparison fair we use only one *SR-Source* for the experiments involving *MPro*.) We use a sample size of 1% of $|Objects|$ for *MPro* and *Upper-Sample*. We report results *without* taking into account the sampling cost and the associated probes for the sample objects, which favors *MPro* and *Upper-Sample* in the comparison. In addition, we performed these experiments over 10,000 queries to be able to report statistically significant results. Figure 17 shows the probing time of the different techniques for different data sets and for the default setting. In all cases, *Upper* performs (slightly) better than *MPro*. In addition, *MPro* has probing times that are similar to those for *MPro-EP*, which also uses query-level probe schedules but does not require object sampling before execution. Using sampling to derive better expected scores helps *Upper-Sample* save probing time with respect to *Upper*. To study the impact of the more expensive local scheduling required by *Upper* and *Upper-Sample*, Figure 18 shows the total processing time of *Upper*

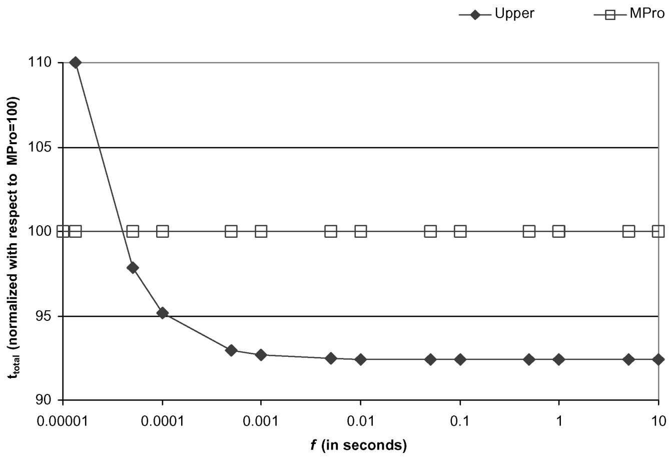


Fig. 18. Total processing time for *Upper* and *MPro*, as a function of the time unit *f*, for the real-life *Cover* data set.

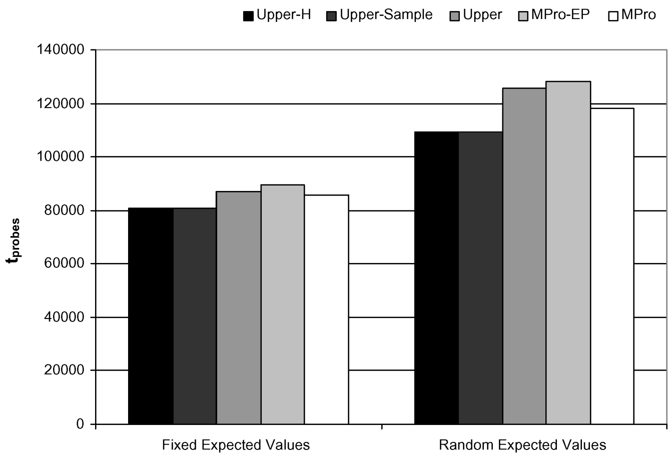


Fig. 19. Performance of *Upper-H*, *Upper-Sample*, *Upper*, *MPro-EP*, and *MPro* for different expected score distributions.

and *MPro* for the real-life *Cover* data set when varying the time unit f, normalized with respect to *MPro*'s total processing time. (The corresponding plots for the other local data sets that we tried show the same trends.) *Upper* is *globally* faster than *MPro* for random access times larger than 0.35 ms (f larger than 0.035 ms). For all configurations tested, with the exception of *Correlated*, *Upper* is faster in terms of t_{probes} time than both *MPro* and *MPro-EP* with a statistical significance of 99.9% according to the t-Test as described by Freedman et al. [1997]. For the *Correlated* data set, *Upper* is faster than *MPro-EP* with a statistical significance of 99.9%, but the difference between *Upper* and *MPro* is not statistically significant.

Figure 19 shows the effect of the source-score distribution on the different techniques when sampling is available. This experiment is similar to that of

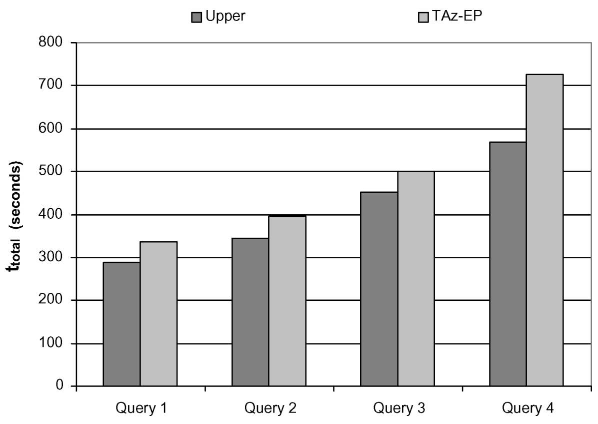


Fig. 20. Experimental results for the real web-accessible data sets relevant to our New York City restaurant scenario.

Figure 16, but only one *SR-Source* is available. In this scenario, *MPro* slightly outperforms *Upper*: *MPro* exploits sampling to characterize the score distribution and determine the scheduling strategy. *Upper-Sample*, which also uses sampling, performs almost as well as the hypothetical *Upper-H* technique. Interestingly, *Upper-Sample* outperforms *MPro* in both experiments. *MPro-EP* has the worst performance of all techniques as it relies on (incorrect) expected values (in the *Rank* metric) and—unlike *Upper*—does not dynamically reevaluate its scheduling choices based on previous probe results.

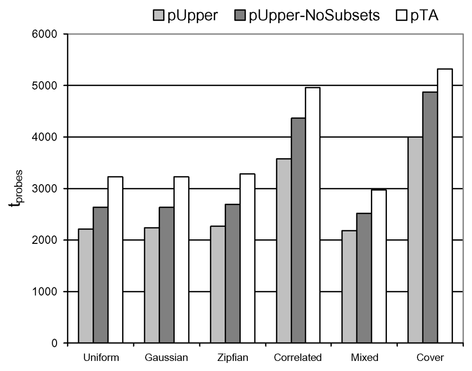
7.1.4 *Real Web-Accessible Data Sets.* Our next results are for the six web-accessible sources, handling 10 attributes, which we described in Section 6.3 and summarized in Table II. To model the initial access time for each source, we measured the response times for a number of queries at different hours and computed their average. We then issued four different queries and timed their total execution time. The source access time is adjusted at run time using the *SRTT* value discussed in Section 6.3.2. Figure 20 shows the execution time for each of the queries, and for the *Upper* and *TA_z-EP* strategies. Because real-web experiments are expensive, and because we did not want to overload web sources, we limited the number of techniques in our comparison. We then focus on our new technique for our web-source scenario, *Upper*, and include *TA_z-EP* as a reasonable "baseline" technique. Just as for the local data sets, our *Upper* strategy performs substantially better than *TA_z-EP*. Figure 20 shows that real-web queries have high execution time, which is a result of accessing the sources sequentially. Parallel versions of the algorithms discussed in this article result in lower overall running times (see Section 7.2). (The *R-Sources* we used are slow, with an average random access time of 1.5 seconds.)

7.1.5 *Conclusions of Sequential Query Processing Experiments.* In summary, our experimental results show that *Upper* and *MPro-EP* consistently outperform *TA_z-EP*: when probing time dominates over CPU-bound probe-scheduling time, interleaving of probes on objects based on the score upper bounds of the objects helps return the top-k query results faster than when we consider objects one at a time. *Upper* outperforms all other techniques—albeit often by a small amount—when no information on the underlying data distribution is known in advance. *MPro-EP* is a very close second and might be an interesting alternative if probes are not too slow relative to local scheduling computation, or for scoring functions where the final object scores cannot be in general approximated or usefully bounded unless all input values are known, as is the case for the *min* function (see Section 4.2). While *Upper*'s dynamic probe scheduling is more expensive in terms of local processing time than *MPro-EP*'s fixed scheduling, the saving in probing time makes *Upper globally* faster than *MPro-EP* in total processing time (although by just a small margin) even for moderate random access times. In addition, *Upper* is *globally* faster that TA_z-*EP* for realistic random access times. In conclusion, *Upper* results in faster query execution when probing time dominates query execution time. When sampling is possible, a variation of *Upper*, *Upper-Sample*, can take advantage of better expected-score estimates, which results in faster query executions. Similarly, *MPro* performs well and adapts better to the data distribution than *MPro-EP* does. Generally, *MPro-EP* (and *MPro* when sampling is possible) are very close in performance to *Upper*, suggesting that the complexity of per-object scheduling of probes (Table III) might not be desirable. However, as we will see in Section 7.2, per-object probe scheduling results in substantial execution-time savings in a parallel processing scenario: per-object scheduling can adapt to intra-query source congestion on the fly, and therefore different probing choices can be made on different objects. 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 article.

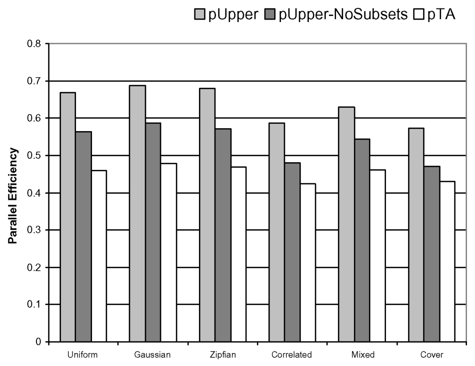
7.2 Parallel Algorithms

In this section, we compare the performance of *pUpper* (Section 5.3) with that of *pTA* (Section 5.2). *pUpper* is a technique in which source probes are scheduled at a fine object-level granularity, and where reevaluation of probing choices can lead objects to be probed in different orders for different sources. In contrast, *pTA* is a technique in which objects are probed in the same order for all sources. In addition, we compare these two techniques with *pUpper-NoSubsets*, a simplification of *pUpper* that does not rely on the *SelectBestSubset* function to make its probing choices. Rather, when a source D_i becomes available, *pUpper-NoSubsets* selects the object with the highest score upper bound among the objects not yet probed on D_i. *pUpper-NoSubsets* is then similar to *pTA*, but with the difference that objects are considered in score-upper-bound order rather than in the order in which they are discovered.

By comparing *pUpper-NoSubsets* and *pTA*, our experiments help identify the saving in probing time that is derived from prioritizing objects on their partial scores. By comparing *pUpper-NoSubsets* and *pUpper*, our experiments



(a) Parallel probing time.



(b) Parallel efficiency.

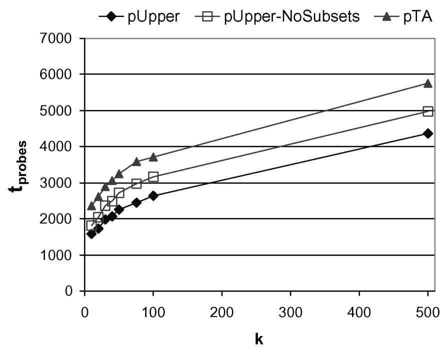
Fig. 21. Effect of the attribute score distribution on performance.

help understand the impact of dynamically selecting probes in a way that accounts for source congestion and known source-score information. These three techniques all react to a source D_i being available to pick a probe to issue next. In Section 7.2.2 we compare these techniques with *Probe-Parallel MPro*, a parallelization of *MPro* presented by Chang and Hwang [2002]. Unlike the other techniques, *Probe-Parallel MPro* does not consider source availability to issue its probes (recall that *MPro* was originally designed for a different setting, namely the execution of expensive predicates for top-*k* queries, not for our web-source scenario) but is based on the concept of "necessary probes," and thus only issues probes that are known to be needed to compute the top-*k* answer.

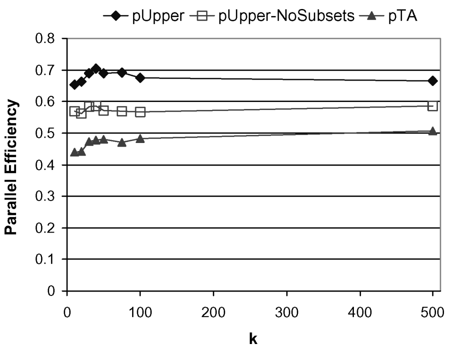
To deploy the *pUpper* algorithm, we first need to experimentally establish a good value for the L parameter, which determines how frequently the random-access queues are updated (Section 5.3). To tune this parameter, we ran experiments over a number of local sources for different settings of $|Objects|$, pR, and k. As expected, smaller values of L result in higher local processing time. Interestingly, while the query response time increases with L, very small values of L (i.e., $L < 30$) yield larger t_{probes} values than moderate values of L (i.e., $50 \leq L \leq 200$) do: when L is small, *pUpper* tends to "rush" into performing probes that would have otherwise been discarded later. We observed that $L = 100$ is a robust choice for moderate to large database sizes and for the query parameters that we tried. Thus, we set L to 100 for the local data experiments.

7.2.1 *Local Data Sets: Performance.* We now report results for the parallel techniques over the local data sets presented in Section 6.2.

Effect of the Attribute Value Distribution. Figure 21 shows results for the default setting described in Table I and for different attribute-value distributions. The probing time t_{probes} of *pUpper*, *pUpper-NoSubsets*, and *pTA* is reported in Figure 21(a). *pUpper* consistently outperforms both *pTA* and *pUpper-NoSubsets*. The dynamic per-object scheduling of *pUpper*, which takes into account source congestion, allows for substantial savings over the simpler *pUpper-NoSubsets* technique. Figure 21(b) shows that *pTA*'s *Parallel Efficiency*



(a) Parallel probing time.



(b) Parallel efficiency.

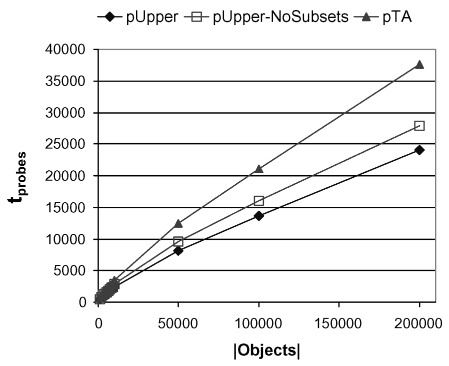
Fig. 22. Effect of the number of objects requested k on performance.

varies slightly among all configurations, with values around 0.45. In contrast, *pUpper*'s *Parallel Efficiency* ranges from 0.57 (*Cover* data set) to 0.69 (*Gaussian* data set), with values of 0.59 for the *Correlated* data sets, 0.63 for the *Mixed* data set, 0.67 for the *Uniform* data set, and 0.68 for the *Zipfian* data set.

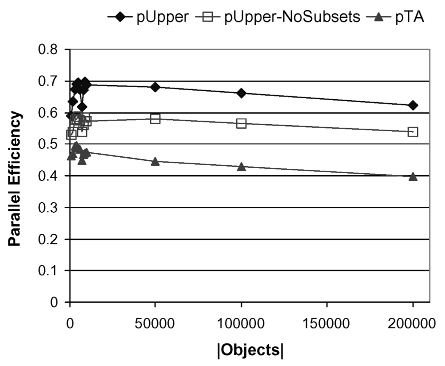
Effect of the Number of Objects Requested k. Figure 22 shows results for the default setting, with t_{probes} and *Parallel Efficiency* reported as a function of k. As k increases, the parallel time needed by *pTA*, *pUpper-NoSubsets*, and *pUpper* increases since all three techniques need to retrieve and process more objects (Figure 22(a)). The *pUpper* strategy consistently outperforms *pTA*, with the performance of *pUpper-NoSubsets* between that of *pTA* and *pUpper*. The *Parallel Efficiency* of *pUpper*, *pUpper-NoSubsets*, and *pTA* is almost constant across different values of k (Figure 22(b)), with *Upper* attaining *Parallel Efficiency* values of around 0.68, which roughly means it is only one third slower than an ideal parallelization of *Upper*.

Effect of the Cardinality of the Objects Set. Figure 23 shows the impact of $|Objects|$, the number of objects available in the sources. As the number of objects increases, the parallel time taken by all three algorithms increases since more objects need to be processed. The parallel time of *pTA*, *pUpper-NoSubsets*, and *pUpper* increases approximatively linearly with $|Objects|$ (Figure 23(a)). The *Parallel Efficiency* of all three algorithms decreases slightly with the number of objects.

Effect of the Number of Parallel Accesses to Each Source $pR(D_i)$. Figure 24 reports performance results as a function of the total number of concurrent random accesses per source. As expected, the parallel query time decreases when the number of parallel accesses increases (Figure 24(a)). However, *pTA*, *pUpper-NoSubsets*, and *pUpper* have the same performance for high $pR(D_i)$ values. Furthermore, the *Parallel Efficiency* of the techniques dramatically decreases when $pR(D_i)$ increases (Figure 24(b)). This results from a bottleneck on sorted accesses: when $pR(D_i)$ is high, random accesses can be performed as soon as objects are discovered, and algorithms spend most of the query processing

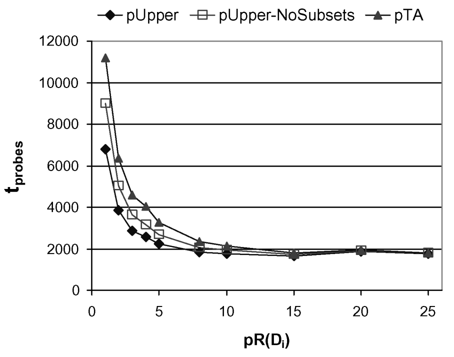


(a) Parallel probing time.

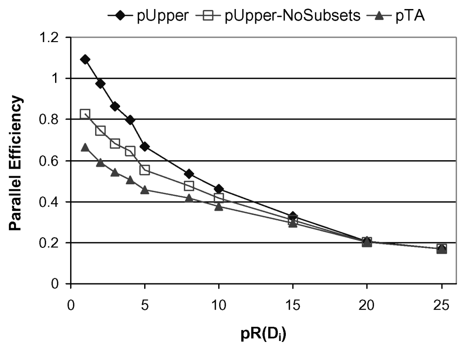


(b) Parallel efficiency.

Fig. 23. Effect of the number of source objects $|Objects|$ on performance.



(a) Parallel probing time.



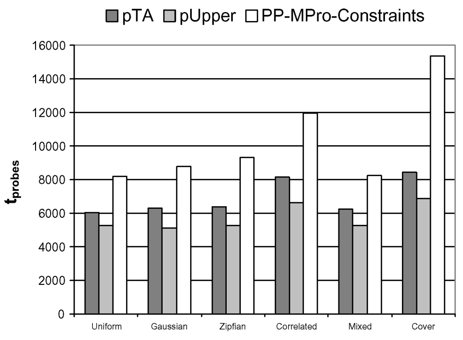
(b) Parallel efficiency.

Fig. 24. Effect of the number of parallel accesses per source $pR(D_i)$ on performance.

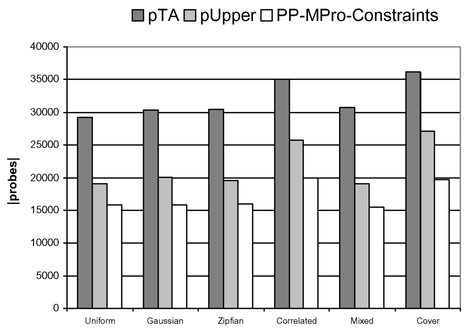
time waiting for new objects to be retrieved from the *SR-Sources*. Surprisingly, for small values of pR, we report *Parallel Efficiency* values that are greater than 1. This is possible since, in the parallel case, algorithms can get more information from sorted accesses than they would have in the sequential case where sorted accesses are stopped as early as possible to favor random accesses; in contrast, parallel algorithms do not have this limitation since they can perform sorted accesses in parallel with random accesses. The extra information learned from those extra sorted accesses might help discard objects faster, thus avoiding some random accesses and decreasing query processing time.

Additional Experiments. We also experimented with different attribute weights and source access times. Consistent with the experiments reported above, *pUpper* outperformed *pTA* for all weight-time configurations tested.

7.2.2 *Local Data Sets: Using Data Distribution Statistics*. If sampling is possible, we can use data distribution information obtained from sampling in the parallel algorithms. In this section, we compare *pUpper* and *pTA* with a



(a) Parallel probing time.



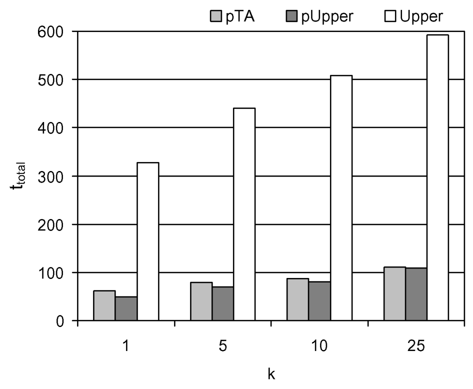
(b) Number of random probes.

Fig. 25. Performance of *pTA*, *pUpper*, and *PP-MPro-Constraints* over different attribute value distributions (one *SR-Source*).

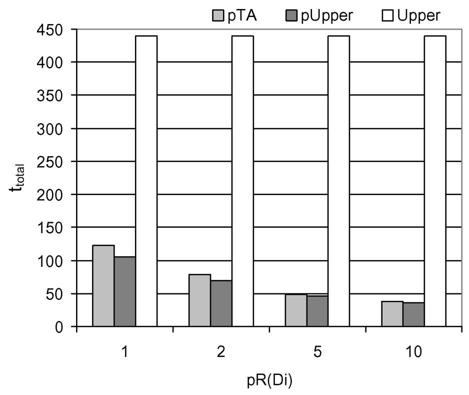
parallelization of the *MPro* algorithm introduced by Chang and Hwang [2002]. For completeness, we also implemented *pUpper-Sample*, a variation of *pUpper* that exploits a sample of the available objects to determine the expected score for each attribute, just as *Upper-Sample* does in the sequential-execution scenario (Section 7.1.3). We observed experimentally that the performance of *pUpper-Sample* is very similar to that of *pUpper*, so for conciseness we do not discuss this technique further.

As mentioned in Section 3, Chang and Hwang [2002] presented a simple parallelization of their *MPro* algorithm, *Probe-Parallel MPro*, which also relies on object sampling to determine its query-level probe schedules. The key observation behind *Probe-Parallel MPro* is that the k objects with the highest score upper bounds all have to be probed before the final top-k solution is found. (Note that this is a more general version of Property 4.1 in Section 4.2.) *Probe-Parallel MPro* simultaneously sends one probe for each of the k objects with the highest score upper bounds. Thus, this strategy might result in up to k probes being sent to a single source when used in our web-source scenario, hence potentially violating source-access constraints. To observe such constraints, we modify *Probe-Parallel MPro* so that probes that would violate a source access constraint are not sent until later. Such a technique, to which we refer as *PP-MPro-Constraints*, does not fully exploit source-access parallelism as some sources may be left idle if they are not among the "top" choices for the k objects with the highest score upper bound. This technique would be attractive, though, for the alternative optimization goal of minimizing the number of probes issued while taking advantage of available parallelism.

Figure 25(a) compares *pTA*, *pUpper*, and *PP-MPro-Constraints* over different data distributions, when only one source provides sorted access. (See our rationale for this setting in Section 7.1.3.) *PP-MPro-Constraints* is slower than the other two techniques because it does not take full advantage of source-access parallelism: a key design goal behind the original *MPro* algorithm is probe minimality. Then, potentially "unnecessary probes" to otherwise idle sources are not exploited, although they might help reduce overall query response time.



(a) Parallel time t_{total} as a function of k ($pR(D_i) = 2$).



(b) Parallel time t_{total} as a function of $pR(D_i)$ ($k = 5$).

Fig. 26. Effect of the number of objects requested k (a) and the number of accesses per source $pR(D_i)$ (b) on the performance of *pTA*, *pUpper*, and *Upper* over real web sources.

Figure 25(b) confirms this observation: *PP-MPro-Constraints* issues on average substantially fewer random-access probes for our data sets than both *pTA* and *pUpper* do. (The three techniques perform approximatively the same number of sorted accesses.) For an alternate optimization goal of minimizing source load, *PP-MPro-Constraints* emerges as the best candidate as it only performs "necessary" probes while still taking advantage of the available parallelism.

7.2.3 *Real Web-Accessible Data Sets.* Our next results are for the real web sources described in Section 6.3.[11] All queries evaluated consider 100 to 150 restaurants. During tuning of *pUpper*, we observed that the best value for parameter L for small object sets is 30, which we use for these experiments.

As in the sequential case (Section 7.1.4), we limited the number of techniques in our comparison because real-web experiments are expensive, and because we did not want to overload web sources. We then focus on the most promising parallel technique for our web-source scenario, *pUpper*, and include *pTA* and *Upper* as reasonable "baseline" techniques. Figure 26(a) shows the actual total execution time (in seconds) of *pTA*, *pUpper*, and the sequential algorithm *Upper* for different values of the number of objects requested k. Up to two concurrent accesses can be sent to each *R-Source* D_i (i.e., $pR(D_i) = 2$). Figure 26(b) shows the total execution time of the same three algorithms for a top-5 query when we vary the number of parallel random accesses available for each source $pR(D_i)$. (Note that pR does not apply to *Upper*, which is a sequential algorithm.) When the number of parallel random accesses to the sources increases, the difference in query execution time between *pTA* and *pUpper* becomes small. This is consistent with what we observed on the local data sets (see Section 7.2.1, Figure 24),

[11]Our implementation differs slightly from the description in Section 6.3 in that we only consider one attribute per source. Specifically, the *Zagat-Review* source only returns the *ZFood* attribute, and the *NYT-Review* source only returns the *TPrice* attribute.

and is due to sorted accesses becoming a bottleneck and slowing down query execution. We also performed experiments varying the relative weights of the different sources. In general, our results are consistent with those for local sources, and *pUpper* and *pTA* significantly reduce query processing time compared to *Upper*. We observed that a query needs 20 seconds on average to perform all needed sorted accesses, so our techniques cannot return an answer in less than 20 seconds. For all methods, an initialization time that is linear in the number of parallel accesses is needed to create the Python subinterpreters (e.g., this time was equal to 12 seconds for $pR(D_i) = 5$). We do not include this uniform initialization time in Figure 26. Interestingly, we noticed that sometimes random access time increases when the number of parallel accesses to that source increases, which might be caused by sources slowing down accesses from a single application after exceeding some concurrency level, or by sources not being able to handle the increased parallel load. When the maximum number of accesses per source is 10, *pUpper* returns the top-*k* query results in 35 seconds. For a realistic setting of five random accesses per source, *pUpper* is the fastest technique and returns query answers in less than one minute. In contrast, the sequential algorithm *Upper* needs seven minutes to return the same answer. In a web environment, where users are unwilling to wait long for an answer and delays of more than a minute are generally unacceptable, *pUpper* manages to answer top-*k* queries in drastically less time than its sequential counterparts.

7.2.4 *Conclusions of Parallel Query Processing Experiments.* We evaluated *pTA* and *pUpper* on both local and real-web sources. Both algorithms exploit the available source parallelism, while respecting source-access constraints. *pUpper* is faster than *pTA*: *pUpper* carefully selects the probes for each object, continuously reevaluating its choices. Specifically, *pUpper* considers probing time and source congestion to make its probing choices at a per-object level, which results in faster query processing and better use of the available parallelism. In general, our results show that parallel probing significantly decreases query processing time. For example, when the number of available concurrent accesses over six real web sources is set to five per source, *pUpper* performs 9 times faster than its sequential counterpart *Upper*, returning the top-*k* query results—on average—in under one minute. In addition, our techniques are faster than our adaptation of *Probe-Parallel MPro* as they take advantage of *all* the available source-access parallelism.

8. CONCLUSION

In this article, 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 sequential source-access scenario. We proposed improvements over existing algorithms for this 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 and schedules probes at the object level, as opposed to other techniques that completely probe one object at a time or do coarser probe scheduling. We showed that probe interleaving greatly reduces query execution time, while the gains derived from object-level scheduling are more modest. The expensive object-level scheduling used in *Upper* is desirable when sources exhibit moderate to high random-access time, while a simpler query-level scheduling approach (such as that used in the *MPro-EP* and *MPro* techniques [Chang and Hwang 2002]) is more efficient when random-access probes are fast. Independent of the choice of probe-scheduling algorithm, a crucial problem with sequential top-*k* query processing techniques is that they do not take advantage of the inherently parallel access nature of web sources, and spend most of their query execution time waiting for web accesses to return. To alleviate this problem, we used *Upper* as the basis to define an efficient *parallel* top-*k* query processing technique, *pUpper*, which minimizes query response time while taking source-access constraints that arise in real-web settings into account. Furthermore, just like *Upper*, *pUpper* schedules probes at a per-object level, and can thus consider intra-query source congestion when scheduling probes. We conducted a thorough experimental evaluation of alternative techniques using both synthetic and real web-accessible data sets. Our evaluation showed that *pUpper* is the fastest query processing technique, which highlights the importance of parallelism in a web setting, as well as the advantages of object-level probe scheduling to adapt to source congestion.

The source-access model on which we base this article, with sorted- and random-access probes, is simple and widely supported by real-web sources. As web services become popular, however, web sources are starting to support more expressive interfaces. Extending the source-access model to capture these more expressive interfaces—as well as developing new query-processing strategies to exploit them—is one interesting direction for future work that we plan to explore.

ACKNOWLEDGMENTS

We thank Kevin Chang and Seung-won Hwang for many fruitful discussions and for helping with our implementation of their *MPro* and *Probe-Parallel MPro* algorithms. We also thank Ron Fagin for his helpful comments on earlier versions of this work.

REFERENCES

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*. ACM, New York.

BLAKE, C. AND MERZ, C. 1998. UCI repository of machine learning databases. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype.

BRUNO, N., CHAUDHURI, S., AND GRAVANO, L. 2002a. Top-*k* selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Datab. Syst. 27*, 2.

BRUNO, N., GRAVANO, L., AND MARIAN, A. 2002b. Evaluating top-*k* queries over web-accessible databases. In *Proceedings of the 2002 International Conference on Data Engineering (ICDE'02)*.

CAREY, M. J. AND KOSSMANN, D. 1997. On saying "Enough Already!" in SQL. In *Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD'97)* ACM, New York.

CAREY, M. J. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB'98)*.

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

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

Chaudhuri, S., Gravano, L., and Marian, A. 2004. Optimizing top-*k* selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng. (TKDE)*.

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

Donjerkovic, D. and Ramakrishnan, R. 1999. Probabilistic optimization of top *n* queries. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB'99)*.

Fagin, R. 1996. Combining fuzzy information from multiple systems. In *Proceedings of the 15th ACM Symposium on Principles of Database Systems (PODS'96)*. ACM, New York.

Fagin, R., Lotem, A., and Naor, M. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS'01)*. ACM, New York.

Fagin, R., Lotem, A., and Naor, M. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 66*, 4.

Freedman, D., Pisani, R., and Purves, R. 1997. *Statistics*. W.W. Norton & Company; 3rd edition.

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

Gravano, L., Marian, A., and Bruno, N. 2002. Evaluating top-*k* queries over web-accessible databases. Tech. Rep., Columbia Univ., New York.

Güntzer, U., Balke, W.-T., and Kießling, W. 2000. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB'00)*.

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

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

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

Mills, D. 1983. Internet delay experiments; RFC 889. In *ARPANET Working Group Requests for Comments*. Number 889. SRI International, Menlo Park, Calif.

Natsev, A., Chang, Y.-C., Smith, J. R., Li, C.-S., and Vitter, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB'01)*.

Nepal, S. and Ramakrishna, M. V. 1999. Query processing issues in image (multimedia) databases. In *Proceedings of the 1999 International Conference on Data Engineering (ICDE'99)*.

Ortega, M., Rui, Y., Chakrabarti, K., Porkaew, K., Mehrotra, S., and Huang, T. S. 1998. Supporting ranked Boolean similarity queries in MARS. *IEEE Trans. Know. Data Eng. (TKDE) 10*, 6, 905–925.

Williams, S. A., Press, H., Flannery, B. P., and Vetterling, W. T. 1993. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.

Received September 2002; revised August 2003; accepted October 2003