Parallel Probing of Web Databases for Top-k Query Processing

Amélie Marian   Luis Gravano

Columbia University
{amelie,gravano}@cs.columbia.edu

Abstract

A "top-k query" specifies a set of *preferred* values for the attributes of a relation and expects as a result the k objects that are "closest" to the given preferences according to some distance function. In many web applications, the relation attributes are only available via *probes* to autonomous web-accessible sources. Probing these sources sequentially to process a top-k query is inefficient, 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. These characteristics of web sources motivate the introduction of *parallel* top-k query processing strategies, which are the focus of this paper. We present efficient techniques that maximize source-access parallelism to minimize query response time, while satisfying source access constraints. A thorough experimental evaluation over both synthetic and real web sources shows that our techniques can be significantly more efficient than previously proposed sequential strategies. In addition, we adapt our parallel algorithms for the alternate optimization goal of minimizing source load while still exploiting source-access parallelism.

1 Introduction

Web search engines usually return the best —or "top k"— matches for a user query. This *top-k query model* is prevalent over multimedia collections in general but also over relational data for applications where users do not expect an exact match for their queries. Top-k queries are a natural choice for applications where users have flexible preferences and tolerate (or even expect) fuzzy matches for their queries. A top-k query then consists of an assignment of target values to the attributes of a relation. To answer such a query, a top-k query processing strategy has to identify the k objects closest to the target values according to some distance function.

Example 1: *Consider a travel site offering last-minute weekend vacation packages. The attributes associated with each package include Origin, Destination, Price, Temperature, and Rating, which correspond, respectively, to the departure city, destination, cost, expected temperature at the destination over the weekend, and average customer rating of the hotel included in the package (e.g., on a scale of 1 to 10). A potential traveler might then specify the departure city, together with preferred values for the Price attribute (e.g., Price=$200), the Temperature attribute (e.g., Temperature=30C), and –perhaps implicitly– the Rating attribute (e.g., Rating=10). (The Temperature value would allow users to express a preference for "warm" destinations, for example.) As a result, the travel site returns, say, the 10 packages for the given departure city that best match the preference specification on Price, Temperature, and Rating, according to some matching function.* ∎

We consider top-k query processing scenarios in which some of the "relation" attributes are handled by remote web sources, and can only be obtained through limited web-accessible interfaces. In the above example, the *Price* attribute could be retrieved from the Orbitz Last-Minute Package web page,[1] which returns a list of vacation packages sorted by price for a given departure city (*sorted access*). The *Temperature* attribute might be available through the AccuWeather web site,[2] which returns the weather forecast for a given location (*random access*). Similarly, the *Rating* attribute might be available through the CitySearch web site[3]. Existing *sequential* algorithms for this top-k query processing scenario attempt to minimize the number of accesses (or *probes*) to the web sources. Unfortunately, any sequential processing strategy for top-k queries over web sources is bound to result in unnecessarily long executions, since web-source accesses may be unreliable and slow due to load and network traffic characteristics.

To radically improve the performance of top-k query processing, in this paper we introduce techniques that fully exploit web-source access parallelism: multiple web

[1] http://packages.orbitz.com
[2] http://www.accuweather.com
[3] http://www.citysearch.com

sources can be accessed simultaneously and, furthermore, individual web sources can typically accept several concurrent accesses at a time. Our top-k query processing strategies then naturally exploit this potential probing parallelism to reduce query response time. A key challenge in the design of these strategies, however, is that sources may pose restrictions on the number of concurrent requests from a single user, to guarantee reasonable response times for all users. Query processing strategies over web sources should then take into account source-access constraints when designing a query execution plan. Furthermore, straightforward adaptations of sequential top-k query processing algorithms to a parallel setting might either not exploit all available parallelism –leaving some sources underutilized– or not adapt dynamically to source congestion –leading to suboptimal source utilization. Some interesting ideas on top-k query parallelization have been recently proposed in the literature [3] (see Sections 4 and 7). However, to the best of our knowledge, our new parallel top-k query processing techniques are the first to be specifically tailored to minimize query response time in the presence of source-access constraints. Our main contributions are as follows:

- We define a realistic source-access model that considers constraints on concurrent accesses that sources might enforce (Section 2).
- We introduce top-k query processing strategies that exploit the inherently parallel access nature of web sources to minimize query response time, while observing source-access constraints (Section 3).
- We present an experimental evaluation of our parallel top-k query processing techniques using both synthetic and real web sources (Section 5). Our parallel techniques manage to achieve close to the maximum theoretical speedup over their sequential counterparts.
- We discuss algorithms for the alternate optimization goal of minimizing source load while still exploiting source-access parallelism.

The rest of the paper is structured as follows: Section 2 defines our query and source model. Section 3 presents our new parallel top-k processing strategies for minimizing response time. Then, Section 4 introduces the data sets and metrics that we use to experimentally evaluate our strategies in Section 5. Section 6 discusses algorithms that aim at minimizing source load rather than query response time. Finally, Sections 7 and 8 review related work and conclude the paper.

2 Background and Problem Statement

The focus of this paper is on parallel query processing techniques for *top-k queries* over web-accessible sources. In this section, we define the top-k query model (Section 2.1) and the source interface that we assume, with its associated access times and source-access constraints (Section 2.2). We also introduce notation (Section 2.3) and the problem that we address in this paper (Section 2.4).

2.1 Query Model

Unlike queries in traditional relational systems, for which the result is a set of tuples, a top-k query returns an *ordered* list of objects, where the ordering is based on how closely each object matches the query. Furthermore, the answer to a top-k query consists only of the k objects that match the query the closest. We use the same query model as in [2], which we review next.

A top-k query over a relation R simply specifies target values for attributes $A_1, \ldots, A_n$ of R. Given a top-k query $q = \{A_1 = q_1, \ldots, A_n = q_n\}$ over a relation R, the score that each 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, which we denote as $\mathit{Score}_{A_i}(q_i, t)$ and assume to be normalized in the $[0, 1]$ range. 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. The final score $\mathit{Score}(q, t)$ for object t is then defined as a weighted sum of the individual scores.[4] The result of a top-k query is the ranked list of the k objects with highest *Score* value, where ties are broken arbitrarily.

2.2 Source Model

Web sources offer several interfaces to access object scores for a given user query. Conceptually, the two most common such interfaces are *sorted access*, which returns a sorted list of objects ranked by score for a given query q, and *random access*, which returns the score of a particular input object for q [5, 6]. The web sources that we consider in this paper can support one or both access interfaces:

Definition 1: [Source Types and Access Time] *Consider an attribute A_i with target value q_i in a top-k query q. Assume further that A_i is handled by a source S. We say that S is an* ***S-Source*** *if, given q_i, we can obtain from S a list of objects sorted in descending order of Score_{A_i} by (repeated) invocation of a* `getNext`(q_i) *probe interface with cost $tS(S)$. (tS stands for "sorted-access time.") Alternatively, assume that A_i is handled by a source R that returns scoring information only 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`(q_i, t) *probe interface, where t is a set of attribute values that identify the object in question, with cost $tR(R)$. (tR stands for "random-access time.") Finally, we say that a source that provides both sorted and random access is an* ***SR-Source****.*

The top-k evaluation strategies that we consider do not allow for "wild guesses" [6]: an object must be "discovered" under sorted access before it can be probed using random access. Therefore, we need to have at least one

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

source with sorted access capabilities to discover new objects. In this paper, we assume that we have one or more *SR-Sources* available, plus arbitrarily many *R-Sources* (see Section 7 for further discussion on this subject).

On the web, sources can typically handle multiple queries in parallel. In this paper, we will produce efficient top-k query processing techniques that exploit this web-source functionality and potentially query each source with multiple probes at a time. However, our techniques must avoid sending large numbers of queries 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 2: [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 up to $pR(S)$ outstanding random-access requests to this same source, for a total of up to $1 + pR(S)$ concurrent accesses.*

2.3 Notation

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 probed the object for some sources but not for 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. $L(t)$ is the corresponding *score lower bound*.

- $E(t)$, 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*, and $\frac{s_\ell(i)}{2}$ if D_i is an *SR-Source*, where $s_\ell(i)$ is the $Score_{A_i}$ score of the last object retrieved from D_i via sorted access. (Initially, $s_\ell(i) = 1$.) [5]

We refer to the set of all objects available through the *SR-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 probe D_i and 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. In this case, we simply use a default value for t's score for attribute A_i.

[5] Several techniques can be used for estimating score distribution (e.g., via sampling) but this topic is beyond the scope of this paper.

2.4 Problem Statement

We consider processing a top-k query over 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. Each source D_i has associated probe times as in Definition 1, and can process at most $pR(D_i)$ concurrent random accesses for the query at any given time, with $pR(D_i) \geq 1$ as in Definition 2. In contrast, since sorted access is sequential by nature, each *SR-Source* can process no more than one sorted access for the query at any given time. We focus on returning the top-k objects for the query as fast as possible. Thus, we will define algorithms that aim at *minimizing the total parallel query processing time, while observing the concurrent-access constraints imposed by each source*. In Section 6 we discuss an alternate cost model in which algorithms aim at minimizing source load rather than query response time.

3 Minimizing Response Time

In this section, we focus on top-k query processing algorithms that attempt to minimize query response time. We first discuss existing algorithms designed for a sequential-processing scenario (Section 3.1). Then, we present our new parallel top-k query processing strategies that observe source-access constraints (Section 3.2).

3.1 Sequential-Processing Scenario

Sequential top-k query processing algorithms 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.

Strategies for a sequential processing scenario differ in their choice of probes. The *TA* algorithm by Fagin et al. [6] retrieves objects for a top-k query via sorted access, and completely probes a retrieved object via random access before probing a new object.[6] The process ends when the

[6] Strictly speaking, the high level description of *TA* in [6] is compatible with implementations in which accesses on multiple sources are allowed to proceed in parallel. In fact, in Section 4.3 we discuss a parallel implementation of *TA* that we use in our experiments in which both sorted and random accesses are allowed to proceed in parallel to minimize query response time.

score of no unseen object can exceed that of the best k objects already seen (which have been fully probed). In summary, when a probe completes, *TA* can either (a) perform a sorted-access probe on a source if the "current" object has been fully probed, or (b) perform a random-access probe on the current object.

The *TA* algorithm completely probes each object that is processed. In contrast, the *Upper* algorithm by Bruno et al. [2] 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 [2]:

Property 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 actual top-k object, then we need to probe all of its attributes to return its final score for q.*
- *If t is not one of the actual top-k objects, its upper bound $U(t)$ is higher than the score of any of the top-k objects. Hence t requires further probes so that $U(t)$ decreases before a final answer can be established.*[7]

Exploiting this property, *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. After the object to probe next is picked, the choice of source to probe for the object is influenced by factors such as the access time and the query weight associated with each source. 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, or (b) select both an object and a source to probe next, guided in both cases by Property 1.

TA and *Upper* are two state-of-the-art processing algorithms for top-k queries. (We discuss others later.) These algorithms do not exploit all the inherent parallelism with which we can access web sources: multiple web sources can be accessed in parallel, and typically web sources accept several concurrent accesses at a time. In the next section, we exploit this observation and present novel parallel processing algorithms for top-k queries. We will focus on adaptations of *Upper* for our parallel scenario, and defer the discussion on *TA* until Section 4.3.

[7]Reference [3] independently presented a top-k query processing algorithm that is based on a similar property. (See Sections 4 and 7 for further discussion.)

3.2 Parallel-Processing Scenario

The focus of this section is on parallel top-k query processing techniques that minimize query response time in the presence of source-access constraints. As explained, 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. Maximizing source-access parallelism helps reduce query processing time. Unlike in the sequential scenario in which strategies decide on object-source pairs to probe, our parallel-scenario strategies choose which object to probe *for the available source*. In this section we first present *pDynamic*, a parallelization of *Upper* for our scenario (Section 3.2.1). As we argue (and show experimentally in Section 5.1), *pDynamic* requires expensive probe scheduling and results in poor overall performance. To address these limitations, we modify *pDynamic* and present *pUpper*, a parallel algorithm that results in efficient query executions while observing source-access constraints (Section 3.2.2).

3.2.1 The pDynamic Strategy

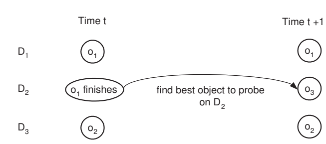


Figure 1: An execution step of *pDynamic*.

As discussed above, 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 upper bound among those objects that need to be probed on D_i according to *Upper*. Figure 1 shows an example execution step of the resulting parallel algorithm, *pDynamic*. In the example, when a source D_2 is done processing a probe on object o_1, we use *Upper*'s probe selection criteria to determine the best object to probe for D_2 (o_3 in this case). We now discuss in detail this new algorithm, which is outlined in Figure 2.

pDynamic is aggressive with respect to sorted accesses: the algorithm attempts to always have exactly one outstanding sorted-access request per *SR-Source* D_i (sorted accesses are sequential by nature; see Definition 2). As soon as a sorted access to D_i completes, a new one is sent until all needed objects are retrieved from D_i (Steps 2-4). When a random access to D_i is available (i.e., when fewer than $pR(D_i)$ outstanding accesses on D_i are being performed), *pDynamic* selects an object (see below) and sends the corresponding random-access probe to D_i (Steps 5-8). Source accesses are performed by calling `pGetNext` (Step 4) and `pGetScore` (Step 8), which are asynchronous versions of the `getNext` and `getScore` source interfaces (Defini-

Algorithm pDynamic (Input: top-k query q)

```
(01) Repeat
(02)     For each SR-Source D_i (1 ≤ i ≤ 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 ≤ i ≤ n):
(06)         While fewer that pR(D_i) random accesses are
             being performed on D_i:
(07)             Select object t to probe for D_i (see text)
(08)             Call pGetScore(D_i, q, t) asynchronously
(09) Until we have identified k top objects
(10) Return the top-k objects along with their scores
```

Figure 2: Algorithm *pDynamic.*

Function SelectBestSubset (Input: object t)

```
(01) Let t' be the object with the k^th largest expected
         score, and let T = E(t')
(02) If E(t) ≥ T:
(03)     Define S ⊆ {D_1, ..., D_n} as the set of all
             sources not yet probed for t
(04) Else:
(05)     Define S ⊆ {D_1, ..., D_n} as the set of
         sources not yet probed for t such that (i) U(t) < T
         if each source D_j ∈ S were to return the expected
         value for t, and (ii) the time Σ_{D_j ∈ S} eR(D_j, t)
         is minimum among the source sets with this property
(06) Return S
```

Figure 3: Function *SelectBestSubset.*

tion 1); these asynchronous calls 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. *pDynamic* terminates when the top-k objects are identified, i.e., when no object can have a final score greater than that of any of the current top-k objects (Step 9).

To select which object to probe next for a source D_i (Step 7 of Figure 2), *pDynamic* uses the *SelectBestSubset* function shown in Figure 3. This 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 3); otherwise (2) only the fastest subset of probes expected to help discard t –by decreasing t's score upper bound below the k^{th} (expected) object score– are considered (Step 5). *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. $eR(D_j, t)$ is 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 $precede(D_j, t)$ denotes the number of such objects, we can define $wR(D_j, t) = \lfloor \frac{precede(D_j,t)}{pR(D_j)} \rfloor \cdot tR(D_j)$.
2. The time $tR(D_j)$ to actually perform the probe.

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

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

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.

In Step 7, *pDynamic* considers the object t with the highest score upper bound (Property 1), and computes its best subset. If an available source D_i is in *SelectBestSubset(t)*, then *pDynamic* sends a random-access probe to D_i for t. Otherwise, the algorithm considers the object with the next highest score upper bound, and so on, until an object that contains D_i in its best subset is found. If no object to probe for D_i is found (which can happen if, for example, all known objects have already been probed for D_i), no probe is sent to D_i and *pDynamic* processes the next source to become available.

The *pDynamic* strategy 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 exponential in the number of sources. To reduce local processing time, we could modify *pDynamic* so that it computes the L next best objects to probe for a source D_i at a time, for a parameter L that regulates the "granularity" at which the algorithm makes decisions. Although this modification would save local processing time, we can devise a more efficient algorithm based on the following observation: whenever *pDynamic* invokes *SelectBestSubset* to schedule probes for a source D_i, the algorithm obtains information on the best probes to perform for D_i *as well as for other sources*. Since *pDynamic* attempts to schedule probes for just one source at any given time, it discards 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. We now show how we can exploit this observation to design an efficient parallel processing algorithm for top-k queries.

3.2.2 The pUpper Strategy

We now present *pUpper*, a new parallel top-k processing algorithm that 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 4. 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

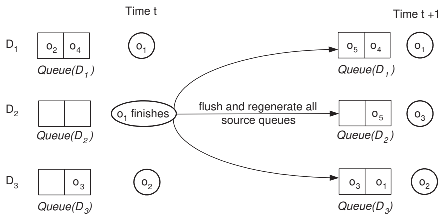


Figure 4: An execution step of *pUpper*.

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

Figure 5: Algorithm *pUpper*.

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 5) associates a queue with each source for random access scheduling. The queues are regularly updated by calls to the function *GenerateQueues* (Figure 6). During top-k query processing, if a source 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 5). If $Queue(D_i)$ is not empty, then simply a probe to D_i on the first object in $Queue(D_i)$ is sent (Step 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*.

To allow for dynamic queue updates at regular intervals, and to ensure that queues are generated using recent information, we define a parameter L, which 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 of Section 3.2.1 (Steps 6-8 in Figure 6). Additionally, as in *pDynamic*, objects are considered in the order of their score upper bound (Step 5).

pUpper precomputes a list of objects to access per

Function GenerateQueues()
(01) Let $Considered$ be the set of objects with score upper bounds greater than the k^{th} largest score lower bound
(02) For each source D_i ($1 \leq i \leq n$):
(03) Empty $Queue(D_i)$
(04) While $Considered \neq \emptyset$ and $\exists i \in \{1, ..., n\} : |Queue(D_i)| < L$:
(05) Extract t_H from $Considered$ such that: $U(t_H) = \max_{t \in Considered} U(t)$
(06) $S = SelectBestSubset(t_H)$
(07) For each source $D_i \in S$:
(08) If $|Queue(D_i)| < L$: $Enqueue(D_i, t_H)$

Figure 6: Function *GenerateQueues*.

source, based on expected value information. Of course, during processing the best subset for an object might vary, 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. We explore values for L experimentally and report our conclusions in Section 5.1.

4 Experimental Setting

In this section, we define the synthetic sources (Section 4.1), real web sources (Section 4.2), techniques we compare (Section 4.3), and metrics (Section 4.4) that we use to evaluate the strategies of Section 3.

4.1 Synthetic 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 three 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 [13].

Correlated: We divided sources into two groups and generated attribute values so that object attributes 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 an object for the other sources in a group are picked 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.

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 maximum number of parallel random accesses $pR(D_i)$ for each source D_i. Table 1 lists the default value for each parameter. Unless we specify otherwise, we use this default setting.

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

Table 1: Default parameter values for experiments over synthetic data.

4.2 Real Web Sources

In addition to experiments over synthetic data, we evaluated our algorithms over real, autonomous web sources. For this, we implemented a prototype of our 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. 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[8]), *DrivingTime* (handled by the MapQuest site[9]), *Popularity* (handled by the AltaVista search engine[10]; see below), *Food* (handled by the Zagat Review web site[11]), and *Price* (provided by the New York Times at the New York Today web site[12]). The Verizon Yellow Pages listing[13], which for sorted access returns restaurants of the user-specified type sorted by shortest distance from a given address, is the only *SR-Source*. We approximate the "popularity" of a restaurant with the number of pages that mention this restaurant as reported by the AltaVista search engine. (This idea of using web search engines as a "popularity oracle" has been used before in the WSQ/DSQ system [7] and in [2].) Attributes *Distance*, *SubwayTime*, *DrivingTime*, and *Food* have "default" target values in the queries (e.g., a *DrivingTime* of 0 and a *Food* rating of 30). The target value for *Popularity* is arbitrarily set to 100 hits, while the *Price* target value is set to the least expensive value in the scale. In the default setting, the weights of all six sources are equal. In a real web environment, the access time tR for a source is usually not constant but rather depends on network traffic. We adapted techniques for estimating round trip time of network packets [10] to develop accurate adaptive estimates for tR.

4.3 Techniques for Comparison

We compare the performance of our new *pDynamic* (Section 3.2.1) and *pUpper* (Section 3.2.2) algorithms against the following three techniques.

***Upper*:** The sequential *Upper* technique [2] (Section 3.1).

***pTA*:** We adapted Fagin et al.'s *TA* algorithm [6] (Section 3.1) for our parallel scenario. The resulting parallel algorithm, *pTA*, probes objects in the order in which they are retrieved from the *SR-Sources*, while respecting source-access constraints. 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* considers optimizations over *TA* so that it can stop probing an object whose score cannot exceed that of the best top-k objects already seen [2].

***MPro-Constraints*:** Chang and Hwang [3] recently presented the *MPro* algorithm, which is based on an independently introduced variation of Property 1. Specifically, their key observation is that the k objects with the highest score upper bounds all have to be probed before the final top-k solution is found. Chang and Hwang propose a parallelization of *MPro*, *Probe-Parallel MPro*, that simultaneously sends one probe for each of the k objects with the highest score upper bounds.[14] Thus, this strategy might result in up to k probes being sent to a single source, hence potentially violating source-access constraints. To observe such constraints, we adapt *Probe-Parallel MPro* so that we block a probe for a chosen object if issuing the probe would violate the access constraints on the associated source.

There are some key differences between *MPro* and *Upper*. Unlike *Upper*, *MPro* uses a "global" probe scheduling where the source-access order is the same for every object. Also, to determine the global schedule, Chang and Hwang resort to an initial object sampling phase. This sampling step is not appropriate for our scenario for two reasons: (1) we cannot easily extract an unbiased, random object sample from the type of web sources that we consider. (*MPro* was designed to optimize the execution of *local expensive predicates*.) Nevertheless, we implemented and experimented with the *MPro* sampling approach over our synthetic sources, and observed that (2) the source ordering guidelines derived via sampling seem not to be effective for weighted-sum scoring functions. Sampling was used primarily in conjunction with *min* as the scoring function in [3], where each individual score is often enough to discard many objects and hence sources can effectively be ordered as a function of their "selectivity." In contrast, the individual selectivity of sources for a weighted-sum scoring function is less useful, as we concluded experimentally. To still be able to compare against (an adaptation of) *MPro*, we replace the sampling-based source scheduling step in [3] with an alternate approach that ranks sources based on their weight/access-time ratio. Finally, note that *MPro* handles only one *SR-Source* (and multiple *R-Sources*), so we restrict our comparison with this technique to this setting. We refer to the resulting parallel technique as *MPro-Constraints*.

4.4 Evaluation Metrics

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

- t_{probes} is the time spent accessing the remote sources.

[8] `http://www.subwaynavigator.com`
[9] `http://www.mapquest.com`
[10] `http://www.altavista.com`
[11] `http://www.zagat.com`
[12] `http://www.nytoday.com`
[13] `http://www.superpages.com`

[14] Chang and Hwang proposed a second parallelization of *MPro* that is not applicable to our web-source setting; see Section 7.

- 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 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:[15]

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

Finally, we quantify the extent to which our parallel techniques exploit the available source-access parallelism. Consider *Upper*, the best sequential algorithm according to the experimental evaluation presented in [2, 8]. 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:

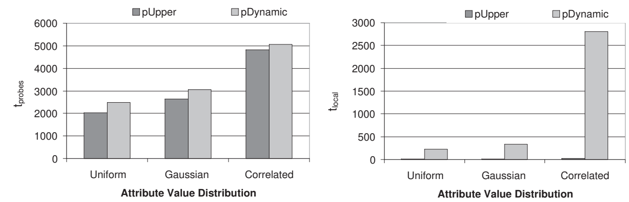
$$Parallel\ Efficiency = \frac{t_{Upper}/(n_{sr} + \sum_{i=1}^{n} pR(D_i))}{t_{probes}}$$

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 synthetic sources, we generate 100 queries randomly. 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 1Ghz 516Mb RAM machines running Red Hat Linux 7.1.

For the real web sources, we defined 12 queries that ask for top French, Italian, and Japanese restaurants in Manhattan, for users located in four different addresses. We report the average t_{total} value for different settings of pR and k. We conducted experiments on a 550Mhz 758Mb RAM machine running Red Hat Linux 7.1.

[15]The number of sorted accesses is more uniform across techniques.



(a) Parallel probing time. (b) Local processing time.

Figure 7: *pUpper* vs. *pDynamic* for the different attribute value distributions.

5 Experimental Results

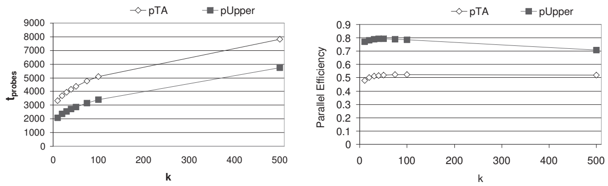
In this section, we present the experimental results for the techniques of Section 3 using the sources and general settings described in Section 4. We first focus on the synthetic sources (Section 5.1), followed by experiments over real web sources (Section 5.2).

5.1 Results for Synthetic Sources

We ran experiments using the synthetic sources of Section 4.1, for various settings of the synthetic-source parameters. We also compared the execution time of the parallel techniques against that of the sequential *Upper* technique and report the corresponding $Parallel\ Efficiency$ values.

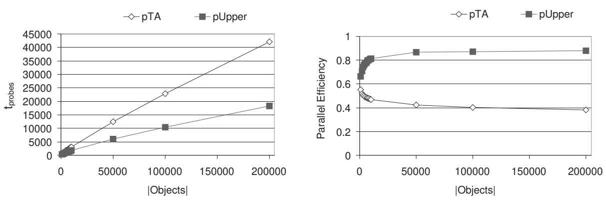
To deploy the *pUpper* algorithm, we first need to experimentally establish a good value for the L parameter, which determines how frequently the random-access queues are updated (Section 3.2.2). To tune this parameter, we ran experiments over a number of synthetic sources for different settings of $|Objects|$, 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 (see discussion on *pUpper* vs. *pDynamic* below). 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 synthetic data experiments.

***pUpper* vs. *pDynamic*:** We experimentally compared *pDynamic* and *pUpper*. Results of this comparison for different data sets are shown in Figure 7. As expected, *pDynamic* is significantly more expensive in terms of local processing time (around 25 times more expensive for the default setting): it requires at least one (and probably many) best-subset computation per source access. By making random-access choices in batches, *pUpper* saves local processing time without harming query response time, since choices are reevaluated frequently. Surprisingly, while we expected *pDynamic* to have lower probing time than *pUpper*, our experiments showed the opposite: *pUpper*'s probing time was 10 to 20% lower than *pDynamic*'s. This can be explained by the greedy nature of *pDynamic*, which tends to choose probes that would have become useless if cur-



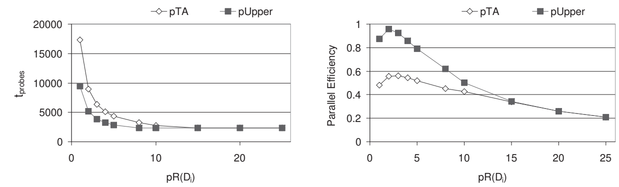
(a) Parallel probing time. (b) Parallel efficiency.

Figure 8: Effect of the number of objects requested k on the performance of *pTA* and *pUpper*.



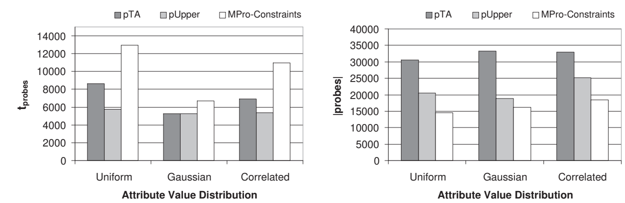
(a) Parallel probing time. (b) Parallel efficiency.

Figure 9: Effect of the number of source objects |*Objects*| on the performance of *pTA* and *pUpper*.



(a) Parallel probing time. (b) Parallel efficiency.

Figure 10: Effect of the number of parallel accesses per source $pR(D_i)$ on the performance of *pTA* and *pUpper*.



(a) Parallel probing time. (b) Number of random probes.

Figure 11: Performance of *pTA*, *pUpper*, and *MPro-Constraints* over different attribute value distributions (one *SR-Source*).

rent outstanding probe results were known. Additionally, *pDynamic* tends to concentrate probes on the first few objects with the highest score upper bounds, whereas *pUpper* considers a wider range of objects. Since experiments on *pDynamic* are very expensive in local processing time and *pUpper* is consistently faster than *pDynamic*, we only consider *pUpper* in the rest of our evaluation. Also, we defer a comparison with *MPro-Constraints*, which as discussed only handles one *SR-Source*, until later in this section.

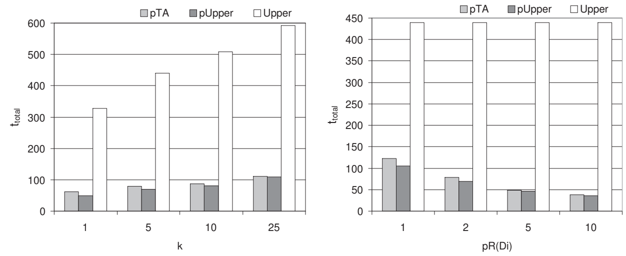
Effect of the Number of Objects Requested k: Figure 8 shows results for the default setting described in Table 1, with t_{probes} and $Parallel\ Efficiency$ reported as a function of k. As k increases, the parallel time needed by *pTA* and *pUpper* increases since both techniques need to retrieve and process more objects (Figure 8(a)). The *pUpper* strategy consistently outperforms *pTA*. *pUpper*'s and *pTA*'s $Parallel\ Efficiency$ decrease when we increase k beyond small values (Figure 8(b)). *pUpper* is particularly efficient for small values of k. For example, when $k = 10$, *pUpper* has a $Parallel\ Efficiency$ value of 0.76, which means it is only 30% slower than the ideal parallelization of *Upper*. When k increases, the best source subsets computed in *pUpper* tend to contain more sources (the k^{th} expected value is low), which leads to smaller savings in terms of random accesses.

Effect of the Number of Source Objects |*Objects*|: Figure 9 shows the impact of |*Objects*|, the number of objects available in the sources. As the number of objects increases, the parallel time taken by both algorithms increases since more objects need to be processed. The parallel time of both *pTA* and *pUpper* increases approximately linearly with |*Objects*| (Figure 9(a)). *pUpper* scales better than *pTA*. Interestingly, *pUpper*'s $Parallel\ Efficiency$ increases with the number of objects, while it decreases for *pTA* (Figure 9(b)): unlike *pTA*, *pUpper* carefully chooses what sources to probe for each object, thus saving random accesses.

Effect of the Number of Parallel Accesses to each Source $pR(D_i)$: Figure 10 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 10(a)). However, *pTA* and *pUpper* have the same performance for high $pR(D_i)$ values. Furthermore, the $Parallel\ Efficiency$ of both techniques dramatically decreases when $pR(D_i)$ increases (Figure 10(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 time waiting for new objects to be retrieved from the *SR-Sources*.

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. We do not discuss these results further because of space limitations. Appendix A reports additional experimental results for varying numbers of sources and attribute-correlation data set configurations.

Comparison with *MPro-Constraints*: Figure 11(a) com-



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

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

Figure 12: 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.

pares *pTA*, *pUpper*, and *MPro-Constraints* over different data distributions, when only one source provides sorted access. (As noted *MPro* was designed to handle only one *SR-Source*.) *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. Figure 11(b) confirms this observation: *MPro-Constraints* issues on average fewer random-access probes for our three data sets than both *pTA* and *pUpper*. The three techniques perform approximatively the same number of sorted accesses. As we discuss in Section 6, *MPro-Constraints* (and adaptations of *TA* and *pUpper*) are good candidates for the alternate scenario in which we attempt to minimize source load, rather than query response time.

5.2 Results for Real Web Sources

Our next results are for the real web sources described in Section 4.2. All queries evaluated consider 100 to 150 restaurants. During tuning of *pUpper* (Section 5.1), we observed that the best value for parameter L for small object sets is 30, which we use for these experiments.

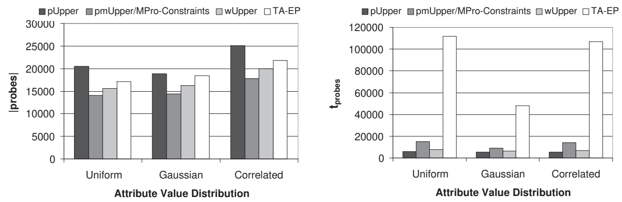
Figure 12(a) shows the actual total execution time (in seconds) of *pTA*, *pUpper*, and the sequential algorithm *Upper* for different values of the number of objects requested k. Up to two concurrent accesses can be sent to each *R-Source* D_i (i.e., $pR(D_i) = 2$). Figure 12(b) shows the total execution 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.) We also performed experiments varying the relative weights of the different sources, which we do not report due to space limitations. In general, our results are consistent with those for synthetic 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 no technique can 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 12. Interestingly, we noticed that sometimes source 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.

Conclusions of Experiments: We evaluated *pTA* and *pUpper* on both synthetic and real-web sources. Both algorithms exploit the available source parallelism, while respecting source-access constraints. 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*.

6 Minimizing Source Load

The main focus of this paper is on minimizing the total parallel query processing time while observing source-access constraints. We now discuss a different optimization scenario, where source load is the minimization objective. In other words, we now attempt to minimize the number of probes that we issue, while still exploiting parallelism whenever possible. Such a scenario would be appropriate for "pay-per-view" sources, or to maximize throughput when many queries are competing for source access. We present some preliminary discussion on how to adapt the algorithms of Section 3 to reduce their required source load, and also report an initial experimental evaluation of the competing strategies.

In the spirit of minimizing source load, all the techniques that we discuss below perform sorted accesses on only one *SR-Source*, and attempt to minimize the number of random accesses for the objects retrieved from the *SR-Source*. If multiple *SR-Sources* are available, one source is arbitrarily chosen for sorted access (e.g., the source with the highest associated query weight). This helps avoid redundant accesses (sorted access on an attribute value that was retrieved via random access). To min-



(a) Number of random probes.

(b) Parallel probing time.

Figure 13: Performance of *pTA*, *pUpper*, and *MPro-Constraints* over different attribute value distributions (one *SR-Source*).

imize the number of sorted-access probes, sorted accesses to the *SR-Source* are stopped as soon as no undiscovered objects can be part of the top-k query answer.

***pmUpper/MPro-Constraints*:** To minimize query response time, *pUpper* aggressively issues probes that might not be strictly necessary to reach a solution. To minimize source load, we adapt *pUpper* so that it only issues probes for the current top-k objects, following the generalization of Property 1 (Section 3.1) in [3]. Furthermore, to favor "high impact" sources, which would hopefully help reduce the number of random probes needed, we do not consider source access time during scheduling. Instead, we use the query weight of the sources as the only criterion for choosing sources. We refer to the resulting algorithm as *pmUpper*. Interestingly, *pmUpper* uses the same random-probe order for each object, since now time (and congestion) are not considered during source-access scheduling. Therefore, *pmUpper* and *MPro-Constraints* become virtually the same algorithm, if we order sources for *MPro-Constraints* just by their query weight (rather than by their associated weight/access-time ratio as before).

***TA-EP*:** The original *TA* algorithm [6] probes each object in turn and decides whether to continue processing a new object based on the (complete) scores of the previously probed objects. Therefore, to strictly minimize the number of probes we need to process objects one at a time. However, rather than completely probing each retrieved object, we can use the shortcut condition presented in [2] for the *TA-EP* variant of *TA*. *TA-EP* is a sequential algorithm that follows the *TA* algorithm but stops probing an object as soon as it can be shown not to be in the query result.

***Experiments*:** Figure 13(a) reports the number of random probes performed by the two source-load minimization algorithms, *pmUpper/MPro-Constraints* and *TA-EP*, against that of *pUpper*, for the experimental setting of Section 5.1 with one *SR-Source* and five *R-Sources*. (*wUpper* is explained below.) As expected, *pmUpper/MPro-Constraints* and *TA-EP* perform fewer random probes than *pUpper*, with *pmUpper/MPro-Constraints* performing fewer probes than any other technique. However, *pmUpper/MPro-Constraints* and *TA-EP* are slower than *pUpper* (Figure 13(b)): the latter algorithm was designed to minimize response time and is therefore able to exploit parallelism more aggressively than the other two algorithms (e.g., the average source utilization for *pUpper* is 95% over the *Correlated* data set, while it is only 25% and 4% for *pmUpper/MPro-Constraints* and *TA-EP*, respectively). *TA-EP* is of course much slower than all other techniques as it accesses sources sequentially. All techniques perform a similar number of sorted accesses.

***wUpper*:** So far, we have discussed query processing techniques that either minimize response time or source load. A simple variation of *pmUpper*, however, exhibits an interesting trade-off between response time and source load: the new algorithm, *wUpper*, considers source congestion during scheduling, and hence incorporates the Section 3.2.1 waiting times when scheduling probes specifically for each individual object. *wUpper* might then not choose the source with the "highest impact" for an object but rather settle for a probe that can be performed immediately. However, *wUpper* only probes objects that will have to be probed (Property 1). By considering source availability in its choices, *wUpper* exhibits low query response times (Figure 13(b)). At the same time, *wUpper* performs only slightly more probes than *pmUpper/MPro-Constraints* (Figure 13(a)). In short, *wUpper* is just a good initial "trade-off" algorithm. A thorough study of how to achieve an appropriate balance of response time and throughput for a specific workload is subject of interesting future work.

7 Related Work

To process top-k queries over multimedia attributes, Fagin et al. proposed a family of algorithms over *SR-Sources* [5, 6]. These algorithms can evaluate top-k queries that involve several independent multimedia "subsystems," each producing scores that are combined using arbitrary monotonic aggregation functions. In an expanded version of [6], Fagin et al. presented a variation of their algorithms to handle *R-Sources*. We discussed adaptations of these algorithms to our parallel access model in Sections 4.3 and 6, and compared them experimentally against our other parallel algorithms.

Nepal and Ramakrishna [11] and Güntzer et al. [9] presented variations of Fagin et al.'s TA algorithm [6] for multimedia query processing. The MARS system [12] also uses variations of the TA algorithm and views queries as binary trees where the leaves are single-attribute queries and the internal nodes correspond to "fuzzy" query operators. Chaudhuri and Gravano built on Fagin's original FA algorithm [5] and proposed a cost-based approach for optimizing the execution of top-k queries over multimedia repositories [4]. Their strategy translates a given top-k query into a selection query that returns a (hopefully tight) superset of the actual top-k tuples.

More recently, Chang and Hwang [3] presented *MPro*, an algorithm to optimize the execution of *expensive predicates* for top-k queries, rather than for our web-source scenario. As such, their "probes" are typically not as expensive as our web-source accesses, hence the need for faster probe scheduling. Unlike *Upper*, *MPro* assumes a fixed schedule of accesses to *R-Sources*, and thus selects which object to probe next but ignores source selection on a per-object basis. 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 adapted this algorithm so that it observes source-access constraints and evaluated it experimentally in Section 6. 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.

Bruno et al. [2] presented *Upper* (Section 3.1) and other sequential algorithms for our top-k query setting, but handled only one *S-Source* (or *SR-Source*) and several *R-Sources*. This restriction is relaxed in [8] to allow for a more flexible scenario of any number of *SR-Sources* and *R-Sources*. Bruno et al.'s original model [2] is a specific instance of this more flexible scenario: when only one *SR-Source* is available, it will only be accessed in sorted access because of the no "wild guesses" restriction. A scenario with several *S-Sources* (with no random-access interface) is problematic: to return the top-k objects for a query together with their scores, as required by our query model, we might have to access *all* objects in some of the *S-Sources* to retrieve the corresponding attribute score for one of the top-k objects. This can be extremely expensive in practice. Fagin et al. presented the *NRA* algorithm [6] to deal with multiple *S-Sources*; however *NRA* only identifies the top-k objects and does not compute their final scores.

The WSQ/DSQ project [7] 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. Finally, Avnur and Hellerstein introduced "Eddies" [1], a query processing mechanism that reorders operator evaluation in query plans. This work shares the same design philosophy as *pUpper*, where we dynamically choose the sources to access next for each object depending on previously extracted probe information.

8 Conclusion

Sequential top-k query processing techniques over web-accessible sources 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. In this paper, we presented efficient *parallel* top-k query processing techniques to minimize query response time while taking source-access constraints that arise in real-web settings into account. We evaluated our new algorithms experimentally using both synthetic and real web sources. Our evaluation showed that our techniques manage to circumvent the high latency of web-source accesses, and perform significantly better than sequential processing strategies in terms of query processing time. In addition, we discussed algorithms for the alternate optimization goal of minimizing source load, and presented preliminary results for this scenario.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*, 2000.

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

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

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

[5] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of the Fifteenth ACM Symposium on Principles of Database Systems (PODS'96)*, 1996.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the Twentieth ACM Symposium on Principles of Database Systems (PODS'01)*, 2001. Expanded version appears on `http://www.almaden.ibm.com/cs/-people/fagin/`.

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

[8] L. Gravano, A. Marian, and N. Bruno. Evaluating top-k queries over web-accessible databases. Technical report, Columbia University, July 2002.

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

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

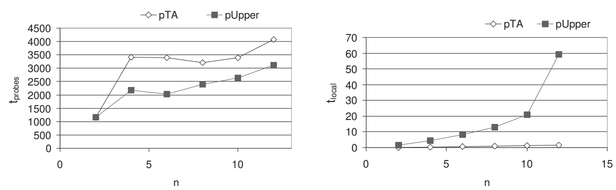
[11] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proc. of the 1999 International Conference on Data Engineering (ICDE'99)*, 1999.

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

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

A Additional Experiments

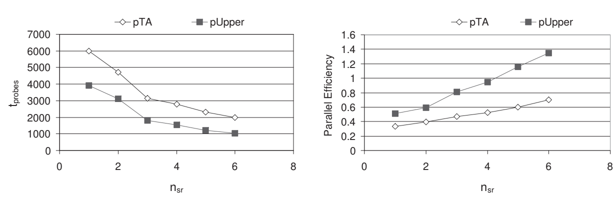
In this appendix, we present additional experimental results for the sources and settings of Section 5.1. Specifically, we report on the effect on the performance of *pTA* (Section 4.3) and *pUpper* (Section 3.2.2) of the number of sources n, the number of *SR-Sources* n_{sr} and *R-Sources* n_r, and the attribute-score correlation .



(a) Parallel probing time. (b) Local processing time.

Figure 14: Effect of the number of sources n on the performance of *pTA* and *pUpper*.

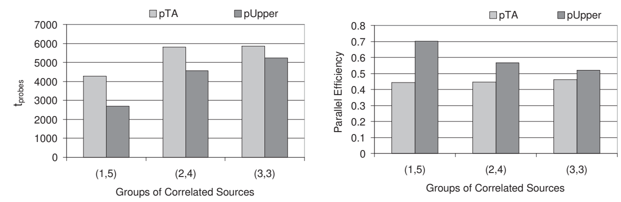
Effect of the Number of Sources n: Figure 14 shows the performance of *pUpper* and *pTA* when we vary the number of sources n. In all cases, we let $n_{sr} = n_r = n/2$. (See below for other values for n_{sr} and n_r.) When $n = 2$, the two algorithms are virtually equivalent. As n increases, the execution time of both algorithms also increases, with *pUpper* outperforming *pTA* (Figure 14(a)). The local processing time of both algorithms is shown in Figure 14(b). *pUpper* is significantly more expensive than *pTA* when n increases: to choose the best sources to probe for an object, *pUpper* takes time that is exponential in the number of sources. However, this is generally acceptable because we expect the number of attributes involved in a top-k query not to exceed a moderate number (e.g., 10).



(a) Parallel probing time. (b) Parallel efficiency.

Figure 15: Effect of the number of *SR-Sources* n_{sr} on the performance of *pTA* and *pUpper*.

Effect of the Number of *SR-Sources* n_{sr} and *R-Sources* n_r: Figure 15 shows the effect of the relative number of *SR-Sources* n_{sr} out of a total of 6 sources on the performance of *pTA* and *pUpper*. When the number of *SR-Sources* increases, the processing time of both algorithms decreases, as more information is obtained from sorted accesses and thus fewer random accesses are needed (Figure 15(a)). Also, we observe an increase in $\mathit{Parallel\ Efficiency}$ for both *pUpper* and *pTA* when n_{sr} increases (Figure 15(b)). Surprisingly, for higher values of n_{sr}, we report $\mathit{Parallel\ Efficiency}$ 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[16].



(a) Parallel probing time. (b) Parallel efficiency.

Figure 16: Effect of attribute value correlation on the performance of *pTA* and *pUpper*.

Effect of Attribute Correlation: To study the effect of attribute-value correlation, we now consider the *Correlated* data sets. Specifically, we divided sources into two groups so that the object values in sources within the same group are correlated. Figure 16 reports the performance of *pUpper* and *pTA* for six sources for three configurations: (1, 5): one group has one source and the other five sources; (2, 4): one group has two sources and the other four sources; and (3, 3): both groups have three sources. Figure 16(a) shows that both algorithms have faster parallel query time for the (1, 5) case, when a large group of five sources is positively correlated. In Figure 16(b), we see that *pUpper*'s $\mathit{Parallel\ Efficiency}$ decreases when sources are split evenly (i.e., for the (3, 3) case), since *pUpper*'s optimizations are less efficient in such a setting. In contrast, *pTA*'s $\mathit{Parallel\ Efficiency}$ is constant among all configurations tested.

[16]Sequential algorithms stop sorted accesses as soon as possible to favor random accesses. Parallel algorithms do not have this limitation since they can perform sorted access 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.