Paper ID 422

Evaluating Top-k Queries over Web-Accessible Databases

Nicolas Bruno   Luis Gravano   Amélie Marian
Computer Science Department
Columbia University
{nicolas,gravano,amelie}@cs.columbia.edu

Abstract

A query to a web search engine usually consists of a list of keywords, to which the search engine responds with the best or "top" k pages for the query. This top-k query model is prevalent over multimedia collections in general, but also over plain relational data for certain applications. For example, consider a relation with information on available restaurants, including their location, price range for one diner, and overall food rating. A user who queries such a relation might simply specify the user's location and target price range, and expect in return the best 10 restaurants in terms of some combination of proximity to the user, closeness of match to the target price range, and overall food rating. Processing such top-k queries efficiently is challenging for a number of reasons. One critical such reason is that, in many web applications, the relation attributes might not be available other than through external web-accessible form interfaces. For example, our food ratings of choice might be provided by a web site that, given a restaurant name, returns the number of "stars" for the restaurant according to the site's critics. To identify the top-10 restaurants for a user query, we will then need to repeatedly query this remote ratings site for a potentially large set of candidate restaurants. In this paper, we study how to process top-k queries efficiently in this setting, where the attributes for which users specify target values might be handled by external, autonomous sources with a variety of access interfaces. We present several algorithms for processing such queries, and evaluate them thoroughly using both synthetic and real web-accessible data.

1 Introduction

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

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

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

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

To process a top-k query over web-accessible databases, we 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

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

and identify the top-10 matches for our query, we then obtain the proximity between each restaurant and the user-specified address by querying MapQuest, and check the average dinner price for each restaurant individually at the NYT-Review source. Hence, we interact with three autonomous sources and repeatedly query them for a potentially large set of candidate restaurants.

Recently, Fagin et al. [7] have presented query processing algorithms for top-k queries for the case where all intervening sources support sorted access (plus perhaps random access as well). Unfortunately, these algorithms are not designed for sources that only support random access (e.g., the MapQuest site in our example above), and such sources abound on the web. In fact, as we will see, simple adaptations of these algorithms do not perform well over random-access sources. In this paper, we present novel processing strategies for top-k queries over sources that support just random access, just sorted access, or both. We also develop non-trivial adaptations of Fagin et al.'s algorithms for random-access sources, and compare these techniques experimentally using a variety of synthetic and real web-accessible data sets.

The rest of the paper is structured as follows. Section 2 reviews relevant work. Section 3 defines our query and data model, and introduces notation and terminology that we use in Section 4 to present our new techniques and our adaptations of Fagin et al.'s algorithms. We evaluate the different strategies experimentally in Section 6 using the data sets and metrics that we outline in Section 5. Finally, Section 7 discusses variations to our query model and highlights interesting directions for future work.

2 Related Work

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

To process queries involving multiple multimedia attributes, Fagin proposed the FA algorithm [6], which was developed as part of IBM Almaden's Garlic project. This algorithm can evaluate top-k queries that involve several independent multimedia "subsystems," each producing scores that are combined using arbitrary monotonic aggregation functions. Fagin showed that this technique is optimal in a probabilistic sense. Recently, Fagin et al. improved on this result and introduced instance-optimal algorithms for the case when all sources provide either both sorted and random access (algorithm TA) or only sorted access (algorithm NRA) [7]. These techniques do not directly handle sources that provide only a random-access interface, which are the focus of our paper. In Section 4.3, however, we adapt Fagin et al.'s algorithms to our scenario. We experimentally compare the resulting techniques with our new approach in Section 6.

Nepal and Ramakrishna [14] and Güntzer et al. [10] presented variations of Fagin's original algorithm [6] for processing queries over multimedia databases. In particular, Güntzer et al. [10] reduce the number of random accesses through the introduction of more stop-condition tests and by exploiting the data distribution. The MARS system [15] also uses variations of the FA algorithm and views queries as binary trees where the leaves are single-attribute queries and the internal nodes correspond to "fuzzy" query operators. Intermediate results are pipelined up the tree structure until they reach the root and are returned to the user. The MARS system can produce results in a demand-driven way, where users ask for the "next best element" for their queries.

Chaudhuri and Gravano also built on Fagin's original FA algorithm and proposed a cost-based approach for optimizing the execution of top-k queries over multimedia repositories [3]. Their strategy translates a given top-k query into a selection query that returns a (hopefully tight) superset of the actual top-k tuples. This approach does not guarantee that the top-k tuples are retrieved by the selection query into which the original top-k query was mapped, and might require repeating the mapping process with a less "selective" predicate. 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 [11, 12]. This technique is then closely related to algorithm TA-EP from Section 4.3.2, which we evaluate experimentally in Section 6.

Over relational databases, Carey and Kossmann [1, 2] present techniques to optimize top-k queries when the scoring is done through a traditional SQL order-by clause. If the scoring function involves multiple attributes, then this technique generally requires an initial scan of the complete relation during query processing. Donjerkovic and Ramakrishnan [5] propose a probabilistic approach to top-k query optimization. This work focuses on relations that might be the result of complex queries including joins, for example, and where the ranking condition involves a single attribute. Finally, Chaudhuri and Gravano [4] exploit multidimensional histograms to process top-k queries over an unmodified relational DBMS by mapping top-k queries into traditional selection queries.

Additional related work includes the general area of information integration, where autonomous sources usually allow only a subset of all queries to be issued over their relations. Halevy et al. [13] introduce *capability records* to model the interface exported by each source, including the limited variable bindings accepted (also called *query templates* in [16]). These capability records are then used to annotate query trees [8] to produce efficient (and valid) execution plans. Our query model can be regarded as some instantiation of this model for top-k queries, where each attribute is handled by exactly one source and where sources provide two kinds of query interfaces, namely sorted and random accesses. Our work in this paper exploits the special characteristics of top-k queries to produce efficient processing strategies for producing ranked query results. Finally, the WSQ/DSQ project [9] presents an architecture for integrating web-accessible search engines with relational DBMSs. The resulting query plans can manage asynchronous external calls to reduce the impact of potentially long latencies. The WSQ/DSQ ideas could be incorporated to our work to speed up the execution of our top-k queries further and depart from the sequential query plans on which we focus in this paper.

3 Query Model

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

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

Consider a top-k query $q = \{A_0 = q_0, A_1 = q_1, \ldots, A_n = q_n\}$ over a relation R. The score that each tuple (or *object*) t in R receives for q is in turn a function of t's score for each individual attribute A_i with target value q_i. Specifically, each attribute A_i has an associated *scoring function Score*$_{A_i}$ that assigns a proximity score to q_i and t_i, where t_i denotes the value of object t for attribute A_i. To combine these individual attribute scores into a final score for each object, each attribute A_i has an associated weight w_i indicating its relative importance in the query. Then, the final score for object t is defined as a weighted sum of the individual scores: [4]

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

where $s_i = \textit{Score}_{A_i}(q_i, t_i)$. The result of a top-$k$ query is then the ranked list of the k objects with the highest *Score* value.

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

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

Recently, techniques have been presented to evaluate top-k queries over traditional relational DBMSs [4, 5]. These strategies assume that all attributes of every object are readily available to the query processor. However, in many applications some attributes might not be available "locally," but rather will have to be obtained from an external web-accessible source instead. For instance, the *Price* attribute in our example is provided by the NYT-Review web site and can only be accessed by querying this site's web interface. Of course, in some cases we might be able to download all this remote information and cache it locally with the query processor. However, this will not be possible for legal or technical reasons for some other sources, or might lead to highly inaccurate or outdated information.

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

Definition 1: [Source Types] *Consider an attribute A_i with target value q_i in a top-k query q. Assume further that A_i is handled by a source S. We will 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`$_S(q_i)$ *interface. Alternatively, assume that A_i is handled by a source R that only returns scoring information when prompted about individual objects. In this case, we will say that R is an R-Source. R provides random access on A_i through a* `getScore`$_R(q_i, t)$ *interface, where t is a set of attribute values that identify an object in question. (As a small variation, sometimes an R-Source will return the actual value of an object for attribute A_i, rather than its associated score.) Finally, we will 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, the Price attribute, from the NYT-Review site, is returned only for specific restaurants (random access). Hence, NYT-Review is an R-Source. Finally, Address is handled by the MapQuest web site, which returns the distance (in miles) between the restaurant address and the user-specified address. Hence, MapQuest is an R-Source.* ▮

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

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

As described in Section 2, Fagin et al. [7] presented query processing algorithms for the case where all sources are either of type *SR-Source* (TA algorithm) or of type *S-Source* (NRA algorithm). As we will see, simple adaptations of these algorithms do not perform as well for the common scenario where *R-Source* sources are also available. In the remainder of this paper, we address this limitation of existing top-k query processing techniques.

4 Evaluating Top-k Queries

In this section we present different strategies for evaluating the top-k queries that we defined in Section 3. Specifically, in Section 4.1 we present a naive but expensive approach to evaluate top-k queries. Then, in Section 4.2 we introduce our novel strategies. Finally, in Section 4.3 we adapt existing techniques designed to solve similar problems to our framework.

For clarity, we make a number of simplifying assumptions in the remainder of this section. We discuss their impact and how we can relax some of them in Section 7. Specifically, we assume that the scoring function for all attributes return values that range between 0 and 1, with 1 denoting a perfect match. Also, we assume that exactly one *S-Source* source (denoted S and associated with attribute A_0) and multiple *R-Source* sources (denoted $R_1, \ldots, R_n$ and associated with attributes $A_1, \ldots, A_n$) are available. (The *S-Source* S could in fact be an *SR-Source* source. In such a case, we will ignore its random-access capabilities in our discussion.) In addition, we assume that only one source is accessed at a time, so all probes are sequential during query processing.

Following Fagin [6, 7], we do not allow our algorithms to rely on "wild guesses": thus a random access cannot zoom in on a previously unseen object, i.e., on an object that has not been previously retrieved under sorted access from a source. Therefore, an object will have to be retrieved from the *S-Source* source before being probed on any *R-Source*. Since we have exactly one *S-Source* S available, objects in S are then the only candidates to appear in the answer to a top-k query. We refer to this set of candidate objects as *Objects*(S). Lastly, we assume that all *R-Source* sources $R_1, \ldots, R_n$ "know about" all objects in *Objects*(S). In other words, given a query q and an object $t \in$ *Objects*(S), we can probe source R_i and obtain the score *Score*$_{A_i}(q_i, t)$ corresponding to q and t for attribute A_i, for all $i = 1, \ldots, n$. Of course, this is a simplifying assumption that is likely not to hold in practice, where each *R-Source* 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. We discuss how to deal with situations like this one in Section 7.

4.1 The Naive Strategy

The simplest technique to evaluate a top-k query q consists of retrieving all partial scores for each object in *Objects*(S), calculating the corresponding combined scores, and finally returning the k objects with the highest scores. We can summarize this procedure as follows:

Algorithm Naive (Input: top-k query q)

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

Using this simple procedure we are guaranteed to return the correct answer to the given top-k query. However, we need to retrieve all scores for each object in *Objects*(S). This can be unnecessarily expensive, especially considering that many scores are not really needed to produce the final answer for the query, as we will see. Using the costs from Definition 2, this strategy takes time $|S| \cdot \left(tS(S) + \sum_{i=1}^{n} tR(R_i) \right)$, where $|S|$ is the number of objects in *Objects*(S).

4.2 Our Proposed Strategies

In this section we present novel strategies to evaluate top-k queries over one *S-Source* and multiple *R-Sources*. Our techniques lead to efficient executions by explicitly modeling the cost of random probes to *R-Sources*. Unlike the naive strategy of Section 4.1, our new algorithms choose *both* the best object and the best attribute on which to probe next at each step of the process. In fact, we will in general not probe all attributes for each object under consideration, but only those attributes that are needed to identify the top-k objects for a query.

Consider an object t that has been retrieved from *S-Source* S and for which we have already probed some subset of *R-Sources* $R' \subseteq \{R_1, \ldots, R_n\}$. Let $s_i =$ *Score*$_{A_i}(q_i, t_i)$ if $R_i \in R'$. (Otherwise, s_i is undefined.) Then, an *upper bound for the score of object* t, denoted $U(t)$, is the maximum possible score that object t can get, consistent with the information from the probes that we have already performed. $U(t)$ is then the score that t would get if t had the maximum score of 1 for every attribute in the query that has not yet been processed for t:

$$U(t) = \textit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n), \text{ where } \hat{s}_i = \begin{cases} s_i & \text{if } R_i \in R' \text{ (i.e., } R_i \text{ has been probed for } t\text{)} \\ 1 & \text{otherwise} \end{cases}$$

If object t has not been retrieved from S yet, then we define $U(t) =$ *ScoreComb*$(s_\ell, 1, \ldots, 1)$, where s_ℓ is the *Score*$_{A_0}$ score for the last object retrieved from S, or 1 if no object has been retrieved yet. (t cannot have a larger score for A_0 than this object, since *S-Source* S returns objects in descending order of *Score*$_{A_0}$.)

Similarly, a *lower bound for the score of an object* t already retrieved from S, denoted $L(t)$, is the minimum possible score that object t can get, consistent with the information from the probes that we have already performed.

$L(t)$ is then the score that t would get if t had the minimum score of 0 for every attribute in the query that has not yet been processed for t:

$$L(t) = \textit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n), \text{ where } \hat{s}_i = \begin{cases} s_i & \text{if } R_i \in R' \text{ (i.e., } R_i \text{ has been probed for } t) \\ 0 & \text{otherwise} \end{cases}$$

If object t has not been retrieved from S yet, then we define $L(t) = 0$.

Finally, the *expected score for an object t* already retrieved from S, denoted $E(t)$, is obtained by assuming that the score for each attribute that has not yet been probed is the expected partial score, which in absence of more sophisticated statistics we set to 0.5:

$$E(t) = \textit{ScoreComb}(s_0, \hat{s}_1, \ldots, \hat{s}_n), \text{ where } \hat{s}_i = \begin{cases} s_i & \text{if } R_i \in R' \text{ (i.e., } R_i \text{ has been probed for } t) \\ 0.5 & \text{otherwise} \end{cases}$$

If object t has not been retrieved from S yet, then we define $E(t) = \textit{ScoreComb}(\frac{s_\ell}{2}, 0.5, \ldots, 0.5)$, where s_ℓ is the $\textit{Score}_{A_0}$ score for the last object retrieved from S, or 1 if no object has been retrieved yet. ($\textit{Score}_{A_0}(q_0, t_0)$ can range between 0 and s_ℓ.)

In Section 4.2.1 we define what constitutes an optimal strategy in our framework. In Section 4.2.2 we describe one new strategy, *Upper*, which can be seen as mimicking the optimal solution when no complete information is available. Finally, in Section 4.2.3 we derive another technique, *Pick*, which aims at greedily minimizing some "distance" between the current state and the final state.

4.2.1 The Optimal Strategy

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

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

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

Using this property, we can view an optimal processing strategy as (a) computing the final scores for the actual top-k objects for a given query, which are needed in the answer, while (b) probing the fewest and least expensive attributes on the remaining objects so that their upper bound is strictly lower than the scores of the top-k objects. This way, an optimal strategy identifies and scores the top objects, while providing enough evidence that the rest of the objects have been safely discarded.

Algorithm Optimal (Input: top-k query q)

```
1. Let $score_k$ be the actual score of the $k^{th}$ best object for top-$k$ query $q$. (Optimal assumes complete knowledge of all object scores.)
2. Get the best object $t$ for attribute $A_0$, with score $s_0$, from S-Source $S$: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
3. If $U(t) < score_k$, return the top-$k$ objects. (No unretrieved object in Objects$(S)$ can have a higher upper bound than $t$.)
4. (a) If object $t$ is one of the actual top-$k$ objects for $q$, probe all R-Sources to compute Score$(q, t)$.
   (b) Otherwise, probe a subset $R' \subseteq \{R_1, \ldots, R_n\}$ such that:
       • After probing every $R_i \in R'$, it holds that $U(t) < score_k$.
       • The cost $\sum_{R_i \in R'} tR(R_i)$ is minimal among the subsets of $\{R_1, \ldots, R_n\}$ with the property above.
5. Get the next best object $t$ from S-Source $S$: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$ and return to step 3.
```

It is important to note that the *Optimal* algorithm is only of theoretical interest and cannot be implemented, since it requires complete knowledge about the scores of the objects, which is precisely what we are trying to obtain to evaluate top-k queries. However, this "strategy" gives a lower bound for the time needed to evaluate a given top-k query by any algorithm that does not involve "wild guesses," as discussed above.

4.2.2 The Upper Strategy

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

We can observe an interesting property:

Property 2: *Consider a top-k query q and suppose that at some point in time we have retrieved some objects from S-Source S and probed some of the R-Sources for these objects. Let $t \in$ Objects(S) be an object with the highest upper bound among all objects in* Objects(S) *(i.e., $U(t) = \max_{t' \in \text{Objects}(S)} U(t')$). 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 will have to probe all of its attributes to return its final score for q.*
- *If t is not one of the actual top-k objects, its upper bound $U(t)$ is higher than the score of any of the top-k objects. Hence t requires further probes so that $U(t)$ decreases before a final answer can be established.*

This property is illustrated in Figure 1 for a top-3 query. In this figure, each object's possible range of scores is represented by a segment, and objects are sorted by their expected score. From Property 1, objects whose upper bound is lower than the lower bound of k other objects cannot be in the final answer. (Those objects are marked with a dashed segment in Figure 1.) Also, the object with the highest upper bound, noted **U** in the figure, will have to be probed before a solution is reached: either **U** is one of the top-3 objects for the query and its final value needs to be returned, or its upper bound will have to be lowered through further probes so that we can safely discard it. Finally, the "current top-k" objects in the figure are those objects with the highest expected score, i.e., those objects whose expected score is no less than a *threshold* that is the k^{th} highest expected score. This threshold will play the role of $score_k$ in algorithm *Optimal*.

Algorithm Upper (Input: top-k query q)

```
1. Get the best object $t$ for attribute $A_0$ from S-Source $S$: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
2. Initialize $U_{unseen} = U(t)$, $Candidates = \{t\}$, and $returned = 0$.
3. Pick $t_H$ from $Candidates$ such that $U(t_H) = \max_{t' \in Candidates} U(t')$.
4. (a) If $U(t_H) < U_{unseen}$:
       • Get the next best object $t$ for attribute $A_0$ from $S$: $(t, s_0) \leftarrow \texttt{getNext}_S(q_0)$.
       • Update $U_{unseen} = U(t)$ and insert $t$ into $Candidates$.
   (b) Else:
       i. If $t_H$ is completely probed:
            • Return $t_H$ with its score; remove $t_H$ from $Candidates$.
            • $returned = returned + 1$. If $returned == k$, halt.
       ii. Else:
            • $Ri \leftarrow SelectBestSource(t_H, Candidates, k - returned)$.
            • Probe source $R_i$ on object $t_H$: $s_i \leftarrow \texttt{getScore}_{R_i}(q_i, t_H)$.
5. Go to step 3.
```

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

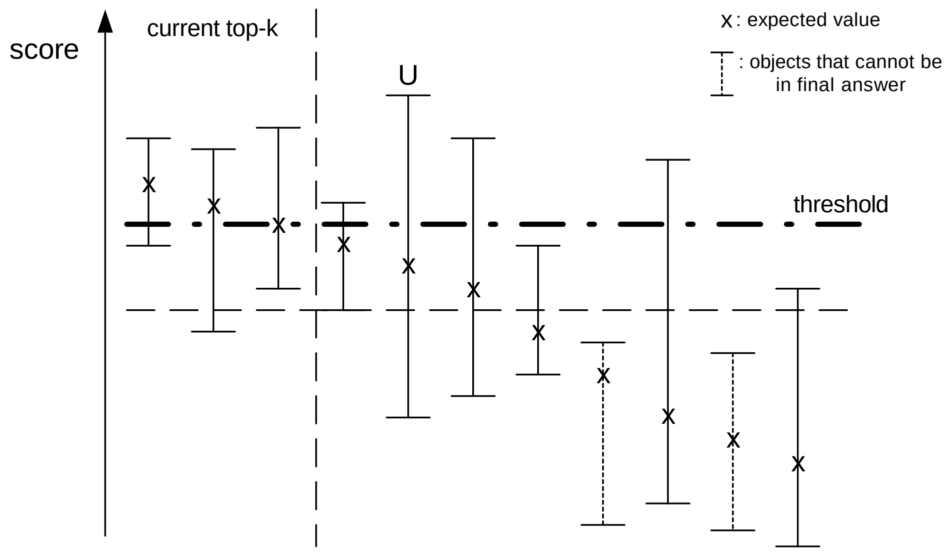


Figure 1: Snapshot of the execution of the *Upper* strategy.

We now discuss how we select the best source to probe for an object t in step 4.b.ii of the algorithm. As in *Optimal*, we concentrate on (a) computing the final value of the top-k objects, and (b) for all other objects, decreasing their upper bound so that it is lower than the scores of the top-k objects. However, unlike *Optimal*, *Upper* does not know the actual scores a-priori and must rely on expected values to make its choices. For an object t, we select the best source to probe as follows. If t is expected to be in the final top-k, i.e., its expected value is one of the k highest ones, we want to compute its final score, and all sources not yet probed for t are considered. Otherwise, we only consider the fastest subset of sources not probed for t that is expected to decrease $U(t)$ under the value of the k^{th} largest expected score (*threshold* T). The best source for t is the one that has the highest $\frac{weight}{cost}$ ratio, i.e., the one that is expected to have a high impact on t's possible score range while being fast:

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

```
1. Let $t'$ be the object in $Candidates$ with the $r^{th}$ largest expected score. Let $T = E(t')$.
   (a) If $E(t) \geq T$:
       • Define $R' \subseteq \{R_1, \ldots, R_n\}$ as the set of all sources not yet probed for $t$.
         ($t$ is expected to be one of the top-$k$ objects, so it needs to be probed on all attributes.)
   (b) Else, define $R' \subseteq \{R_1, \ldots, R_n\}$ so that:
       • $U(t) < T$ if each source $R_i \in R'$ were to return the expected value for $t$, and
       • The cost $\sum_{R_i \in R'} tR(R_i)$ is minimal.
       (Since $E(t) < T$, we are guaranteed to find at least one such set of attributes.)
2. Return a source $R_i \in R'$ such that $\frac{w_i}{tR(R_i)}$ is maximum (i.e., we favor fast probes).
```

4.2.3 The Pick Strategy

We now present the *Pick* algorithm, which uses a different approach to evaluate top-k queries. While *Upper* chooses the probe that is most likely to be in the "optimal" set of probes, *Pick* chooses the probe that minimizes a certain function B, which represents the "distance" between the current state and the final state, in which the top-k tuples are easily extracted. At a given point in time in the execution, function B focuses on t', the object with the k^{th} highest expected score among the objects retrieved from *S-Source* S. Furthermore, B considers the range of possible scores that each such object can take above $E(t')$. The smaller such ranges are, the closer we are likely to be to finding the final solution for the query. In effect, when we reach the final state, t' is the object with the actual k^{th} highest score, and all objects not in the answer should be known *not* to have scores above that of t'. The definition of function B then becomes:[5]

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

Figure 2 shows a snapshot of a query execution step, highlighting the score ranges that "prevent" the current state from being the final state. Note that the value of B is never negative. When B becomes zero, all top-k scores are known, and all objects not in the final answer have an upper bound for their score that is lower than $E(t')$.

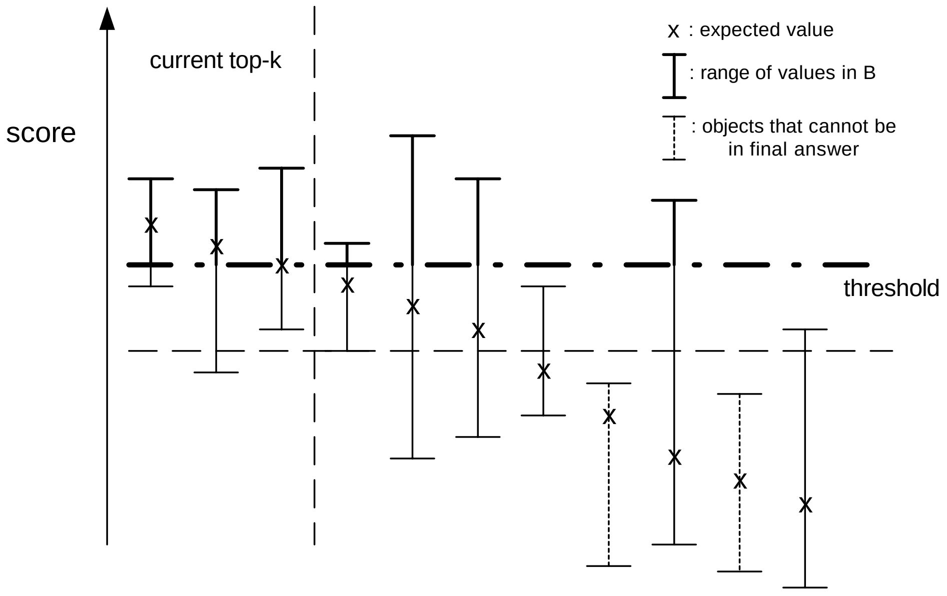


Figure 2: Snapshot of the execution of the *Pick* strategy.

At each step, *Pick* greedily chooses the probe that would decrease B the most in the shortest time. *Pick* selects for each object the best source to probe, i.e., the attribute value that will result in the highest decrease in B. If

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

the object is expected to be in the final top-k answer, all unprobed attributes are considered. Otherwise, only attributes from the best (fastest) set of attributes that will be needed to eliminate the object are considered. This is completely analogous to how the *SelectBestSource* function works. Then, among all selected object-*R-Source* pairs, *Pick* chooses the one with the highest $\frac{expected\ decrease\ of\ B}{tR(R)}$ ratio.

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

Algorithm Pick (Input: top-k query q)

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

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

4.3 Existing Approaches

While existing algorithms in the literature are designed for different scenarios, e.g., they assume that all sources are *SR-Sources* (TA), we can adapt them to our framework and use them for comparison purposes. In Section 4.3.1 we adapt the TA algorithm [7] so that it also works over *R-Sources*, and in Section 4.3.2 we extend the resulting algorithm so that it also incorporates ideas from the expensive-predicates literature. As an important difference with our strategies of the previous section, all the techniques presented below choose an object and probe all needed sources before moving to the next object. This coarser strategies can degrade the overall efficiency of the techniques, as shown in Section 6.

4.3.1 Fagin's Algorithms

Fagin et al. [7] presents the TA algorithm for processing top-k queries over *SR-Sources*. This strategy is proven to be instance optimal [7] over all algorithms that do not perform wild guesses:

Algorithm TA (Input: top-k query q)

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

Although this algorithm is not designed to deal with *R-Source*s, we can adapt it in the following way. In step 1, we access the only *S-Source* S using sorted access, and retrieve an object t. In step 2, we define the threshold value as $U(t)$, since the maximum possible score for any *R-Source* is always 1. Then, for each object t retrieved from S we probe all *R-Source*s to get the final score for t. For a model with a single *S-Source* S, the modified algorithm retrieves in order all objects in *Objects*(S) one by one and determines whether each object is in the final answer by probing all the remaining *R-Source*s. The complete procedure is described below.

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

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

We can improve the algorithm above by interleaving the execution of steps 3 and 4 and adding a shortcut test condition. Given an object t, we calculate the value $U(t)$ after each random probe to an *R-Source* R_i, and we skip directly to step 5 if the current object t is guaranteed not to be one of the top-k objects. That is, if $U(t)$ is lower than the lowest score of the current top-k objects, we can safely ignore object t (see Property 1) and continue with the next one. We call this algorithm *TA-Opt*.

4.3.2 Exploiting Techniques for Processing Selections with Expensive Predicates

Work on expensive-predicate query optimization [11, 12] has studied how to process selection queries of the form $p_1 \wedge \ldots \wedge p_n$, where each predicate p_i can be expensive to calculate. The key idea is to order the evaluation of predicates to minimize the expected execution time. The evaluation order is determined by the predicates' *rank*, defined as:

$$rank_{p_i} = \frac{selectivity(p_i)}{cost\text{-}per\text{-}object(p_i)}$$

where $cost\text{-}per\text{-}object(p_i)$ is the average time to evaluate predicate p_i over an arbitrary object.

We can adapt this idea to our framework in the following way. Let $R_1, \ldots, R_n$ be the *R-Sources*, and let $w_1, \ldots, w_n$ be the corresponding weights in the *Score* function. We sort the *R-Sources* R_i in increasing order of rank, defined as:

$$rank_{R_i} = \frac{w_i \cdot E(Score_{R_i})}{tR(R_i)}$$

where $E(Score_{R_i})$ is the expected score of an object from source R_i (typically 0.5 unless we have more sophisticated statistics). Thus, we favor fast sources that might have a greater impact on the final score of an object, i.e., those sources that are likely to significantly change the values of $U(t)$ and $L(t)$.

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

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

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

5 Evaluation Setting

In this section we describe the data sets (Section 5.1) and metrics and other settings (Section 5.2) that we use to evaluate the strategies of Section 4.

5.1 Data Sets

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

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

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

- *"Gaussian"* data set: We generate the multiattribute score distribution by producing five overlapping multi-dimensional Gaussian bells [17].

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

Real Web-Accessible Sources: The real sources that we use are relevant to (an expanded version of) our running example of Section 3. Users input a starting address, the type of cuisine in which they are interested (if any), and importance weights for the following *R-Source* attributes: *SubwayTime* (handled by the SubwayNavigator site [6]), *DrivingTime* (handled by the MapQuest site), *Popularity* (handled by the AltaVista search engine [7]; see below), *ZFood*, *ZPrice*, *ZDecor*, and *ZService* (handled by the Zagat Review web site), and *TRating* and *TPrice* (provided by the New York Times at the New York Today web site). The Verizon Yellow Pages listing [8], which returns restaurants of the user-specified type sorted by shortest distance from a given address, is the only *S-Source*. Table 1 summarizes the real sources in our setting, together with details on their associated interface.

The *Popularity* attribute requires further explanation. We approximate the "popularity" of a restaurant with the number of web pages that mention the restaurant, as reported by the AltaVista search engine. (The idea of using web search engines as a "popularity oracle" has been used before in the WSQ/DSQ system [9].) Consider, for example, restaurant "Tavern on the Green," which is one of the most popular restaurants in the United States. A query on AltaVista on "Tavern on the Green" AND "New York" returns 1,972 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 six 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 popularity oracles in most cases.

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

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

Table 1: The real web-accessible sources used in the experimental evaluation of the various techniques.

Of course, the real sources above do not fit our model of Section 3 perfectly. For example, some of these sources return values for multiple attributes simultaneously (e.g., the Zagat Review site). Also, as we mentioned before, information on a restaurant might be missing in some of these sources (e.g., a restaurant might not have an entry at the Zagat Review site). In such a case, our system will give a default (expected) value to the score of the corresponding attribute. We address these issues in Section 7.

5.2 Other Experimental Settings

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

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

where n_S is the number of objects extracted from *S-Source* S, n_i is the number of random-access probes for *R-Source* R_i, and tS and tR are as specified in Definition 2. t_{total} then approximates the total execution time for a query. Where appropriate, we report t_S and t_R, the sorted- and random-access components of this time, respectively, together with the actual number of sorted (n_S) and random accesses ($n_R = \sum_{i=1}^{n} n_i$) required. Finally, $n_{total} = n_S + n_R$.

For the synthetic data sets and for each setting of the experiment parameters, we generate 100 queries randomly, with their associated weights, produce costs and attribute values for the sources, and compute the average t_{total}, t_S, t_R, n_{total}, n_S, and n_R values. We report results for top-k queries for different values of k, $|S|$, cf and for various assignments of weights and costs to sources. In the default setting, k is 10 (i.e., queries ask for the best 10 objects), $|S| = 5,000$, and we use the *Uniform* data set.

Query	k	*Address*	*Cuisine*	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9
1	5	112th Street	Any	10	5	5	5	1	1	1	1	1	1
2	5	73rd Street	Any	10	5	5	5	1	1	1	1	1	1
3	5	112th Street	Chinese	10	5	5	5	1	1	1	1	1	1
4	5	112th Street	French	10	5	5	5	1	1	1	1	1	1
5	5	73rd Street	Italian	10	5	5	5	1	1	1	1	1	1
6	10	73rd Street	Italian	10	5	5	5	1	1	1	1	1	1
7	5	73rd Street	French	10	1	2	5	4	0	0.2	0.2	1	3

Table 2: The seven queries for the experiments over our real data sets, including the weight for each attribute (A_0 =*Distance*, A_1 =*SubwayTime*, A_2 =*DrivingTime*, A_3 =*Popularity*, A_4 =*ZFood*, A_5 =*ZPrice*, A_6 =*ZDecor*, A_7 =*ZService*, A_8 =*TRating*, and A_9 =*TPrice*).

For the real data sets, we use seven queries, some specifying an address on E. 73rd Street, and some others specifying an address on W. 112th Street. Attributes *Distance*, *SubwayTime*, *DrivingTime*, *ZFood*, *ZDecor*, *ZService*, and *TRating* have "default" target values in the queries (e.g., a *DrivingTime* of 0 and a *ZFood* rating of 30). The target value for *Popularity* is 1,000 hits, while *ZPrice* and *TPrice* are set to the least expensive value in the scale. In all seven queries, the weight of the *S-Source* attribute (i.e., *Distance*) is roughly twice the weight of any *R-Source* attribute. See Table 2 for further details.

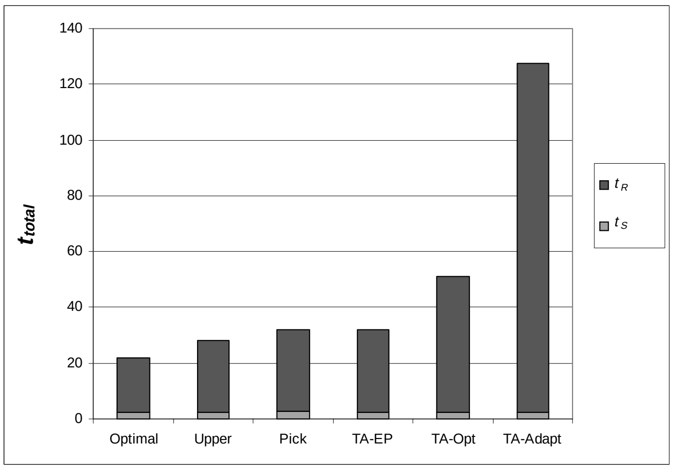
Next, we experimentally compare the algorithms that we discussed in Section 4, namely *TA-Adapt* (Section 4.3.1), *TA-Opt* (Section 4.3.1), *TA-EP* (Section 4.3.2), and our novel strategies, *Upper* (Section 4.2.2) and *Pick* (Section 4.2.3). We also report results for the *Optimal* technique of Section 4.2.1. As discussed, this technique is only of theoretical interest, and serves as a lower bound for the time that any strategy without "wild guesses" would take to process a given top-k query.

6 Evaluation Results

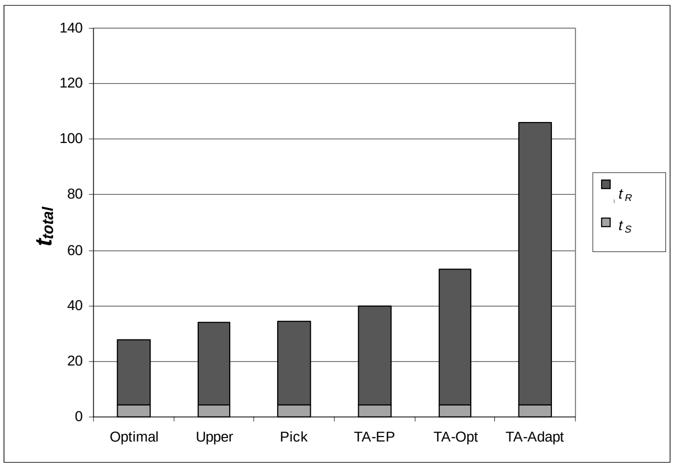
In this section we present the experimental results for the techniques of Section 4 using the data sets and general settings described in Section 5.

6.1 Results for Synthetic Data Sets

The first results that we report are for the *default setting* of the experiment parameters (Section 5). Figure 3 shows the average execution time for each technique and for both the *Uniform* and *Gaussian* synthetic data sets. The *Upper* strategy consistently outperforms all other techniques, and has total execution time close to that of the lower bound,



(a) *Uniform data set*



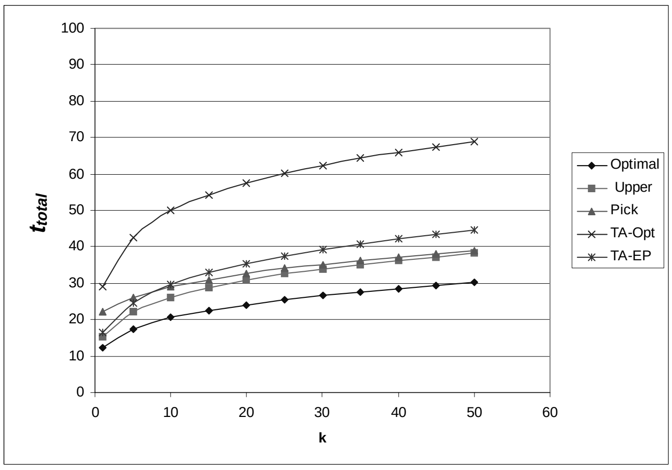
(b) *Gaussian data set*

Figure 3: Performance of the different strategies for the default setting of the experiment parameters, and for two synthetic data-set distributions.

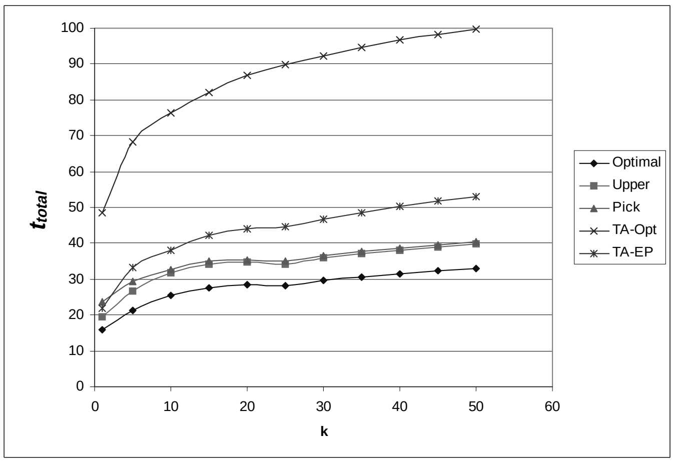
Optimal. For *Uniform*, *Pick* and *TA-EP* exhibit similar overall execution times. However, *Pick* performs better than *TA-EP* for *Gaussian*. *Pick* gives better results for *Gaussian* than for *Uniform* because the *Gaussian* distribution allows *Pick* to retrieve fewer objects from *S-Source* S at initialization, which in turn results in fewer objects being subjected to random-access probes. All techniques need the same number of sorted accesses, except *Pick*, which does slightly more accesses to *S-Source* S since it initially needs to retrieve a large number of objects under sorted access (Section 4.2.3). We can see that our optimizations over *TA-Adapt*, namely *TA-Opt* and *TA-EP*, result in dramatic improvements in performance over *TA-Adapt*. We then remove *TA-Adapt* from further consideration in the remaining discussion.

Effect of the Number of Objects Requested k: We study the effect of varying the value of k in Figure 4. We report results for the default setting, as a function of k and for both the *Uniform* and *Gaussian* synthetic data sets. As k increases, the time needed by each algorithm to return the top-k objects increases as well, since all techniques need to retrieve and process more objects. The general drop in performance is substantial from $k = 1$ to $k = 10$ but is less pronounced for higher values of k. The relative performance of the different techniques remains similar for a wide range of values of k. When k increases, *Pick* performs better than *TA-EP*, especially for *Gaussian*. For $k > 10$, *Upper* and *Pick* show similar results. Observe that the difference between the execution time of *Upper* and *Optimal* is almost constant (and small) across different k values.

Effect of the Number of Objects in *S-Source* S**:** Figure 5 studies the impact of the size of *S-Source* S. For each size, we report the average execution time for 50 randomly generated queries. 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



(a) *Uniform data set*



(b) *Gaussian data set*

Figure 4: Performance of the different strategies for the default setting of the experiment parameters, as a function of the number of objects requested k, and for two synthetic data-set distributions.

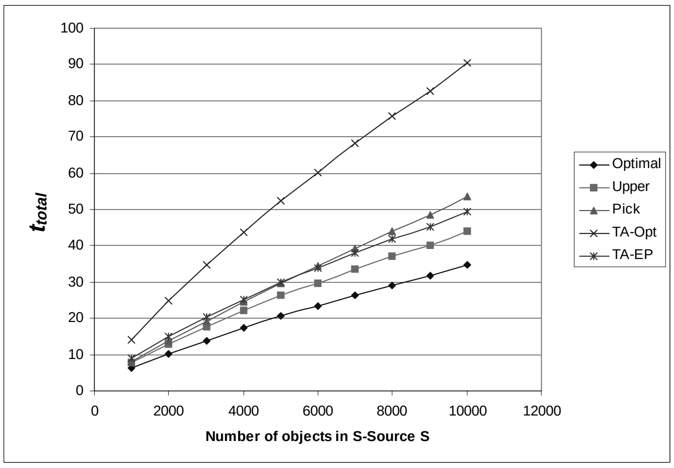
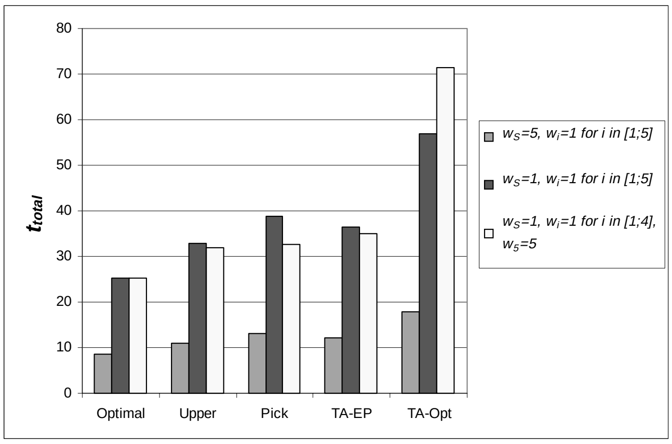


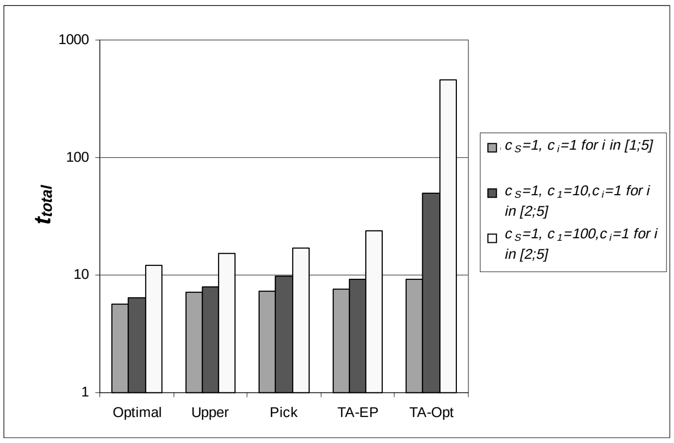
Figure 5: Performance of the different strategies for the default setting of the experiment parameters, as a function of the number of objects in *S-Source* S.

time needed by each algorithm is approximately linear in the number of objects in S. *Upper* gives better results and scales better than other techniques.

Effect of Attribute Weights and Source Costs: We now report on the impact of attribute weights and source access costs on the execution times. In particular, Figure 6(a) shows results for various assignments of weights to the query attributes. (All other parameters of the experiments follow the default setting.) Also, Figure 6(b) shows results for various combinations of costs for the various sources. The results in Figure 6(b) are presented in a logarithmic scale. *TA-Opt* performs poorly when access to one of the sources is significantly more expensive than to the other sources (see Figure 6(b) for $c_1 = 100$). In contrast, *TA-EP*, which orders accesses to *R-Sources* taking access costs into account, performs dramatically better.



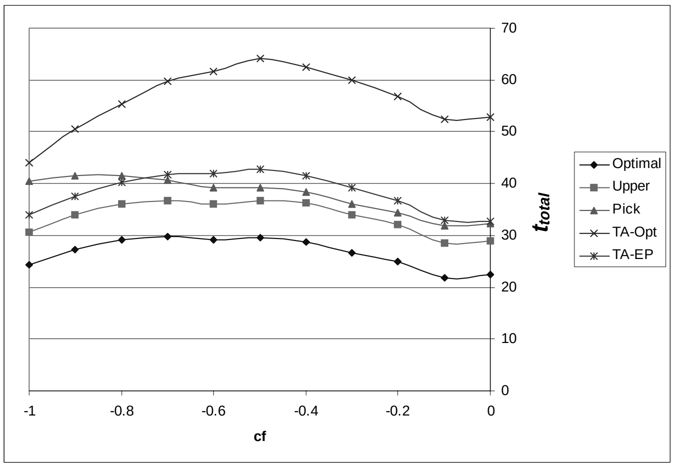
(a) Effect of attribute weights



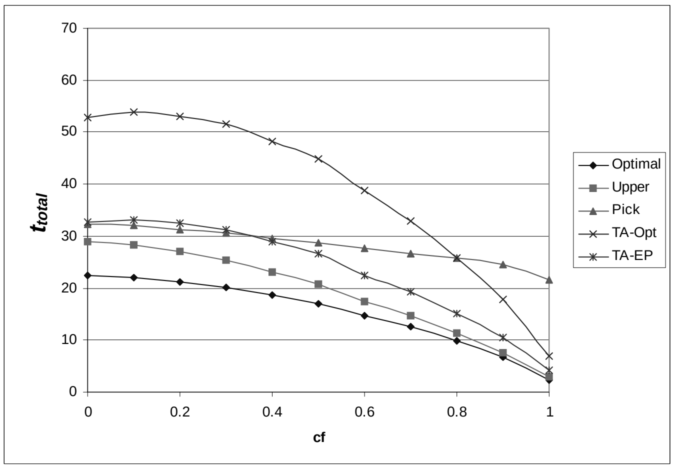
(b) Effect of source access costs (logarithmic scale)

Figure 6: Performance of the different strategies for various attribute-weight and source access-cost combinations.

Effect of Attribute Correlation: We now turn to the *Correlation* data set (Section 5) and evaluate the effect that attribute correlation has on the performance of the query processing techniques. Figure 7(a) is for the case when all *R-Sources* are negatively correlated with *S-Source* (i.e., when correlation factor *cf* < 0; see Section 5). In contrast, Figure 7(b) is for the positive-correlation case (i.e., *cf* > 0). As seen in Figure 7(b), when *cf* is high the performance of all techniques, with the exception of *Pick*, improves dramatically: All techniques other than *Pick* access *S-Source* S on demand and when needed. In contrast, *Pick* extracts a batch of objects during initialization, which results in a substantial number of candidate objects in need of random probes. This way, *Pick* does not benefit from all attribute scores being positively correlated as the other techniques do. Interestingly, a negative correlation between the *R-Sources* and the *S-Source* attribute scores does not affect the performance of the algorithms significantly.



(a) Negative attribute correlation



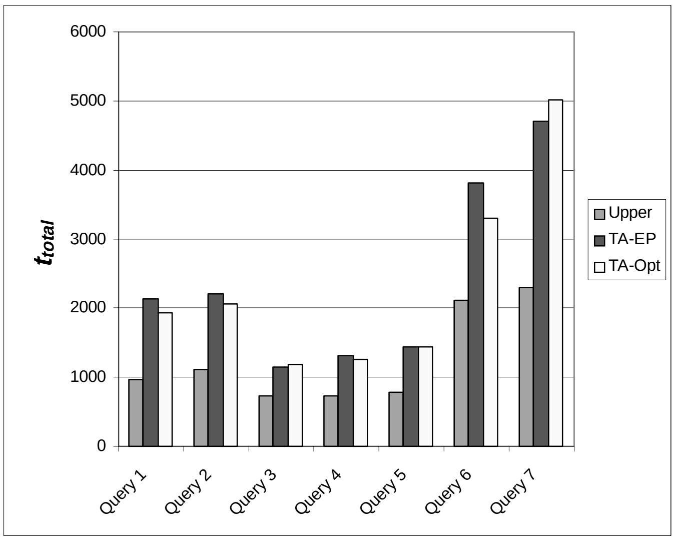
(b) Positive attribute correlation

Figure 7: Performance of the different strategies for the *Correlation* synthetic data set and the default setting of the experiment parameters, as a function of the correlation factor *cf*.

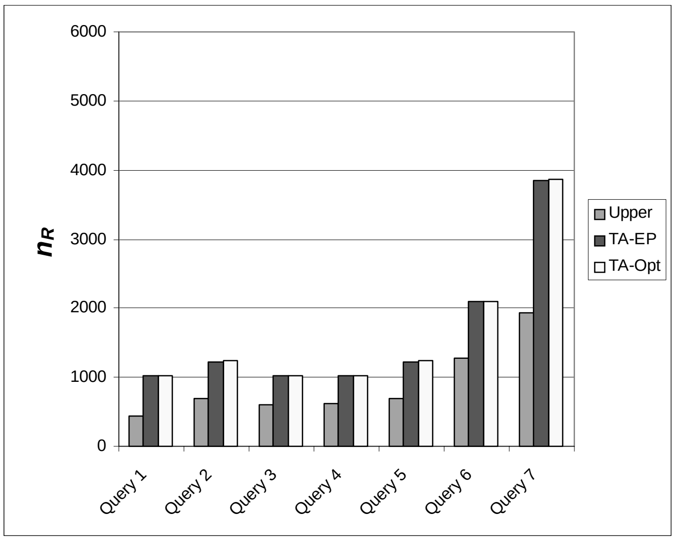
6.2 Results for Real Web-Accessible Data Sets

Our final set of results are for the real data sets that we described in Section 5 and summarized in Table 1. There are six web-accessible sources, handling 10 attributes. To model the access cost for each source, we measured the response time for a number of queries and computed their average. We then issued the seven queries in Table 2 to these sources and timed their execution. Figure 8(a) shows the execution time for each of the queries, and for the *Upper*, *TA-EP*, and *TA-Opt* strategies. Since *Upper* gives consistently better results than *Pick* in the synthetic data experiments, we choose to focus on it in the real data experiments. We compare our *Upper* technique performance with *TA-EP* and *TA-Opt* results, and ignored *TA-Adapt*, whose results in the synthetic data experiments were significantly worse than those for other techniques. Figure 8(b) shows the number of random-access probes that each technique requires for each query. In contrast with the synthetic-data results, *TA-EP* does not outperform *TA-Opt*. We conjecture that this discrepancy is due to our rough estimates for the source access costs, to which the *TA-EP* strategy would be particularly sensitive. In general, just as we observed for the synthetic data sets, our *Upper* strategy performs significantly better than the two versions of the *TA* algorithm that we tried.

In summary, our experimental results consistently show that *Upper* outperforms all other methods, with performance close to that of the *Optimal* technique, for both synthetic and real data sets. Furthermore, our modifications to the *TA* algorithm, *TA-EP* in particular, resulted in significant improvements in performance.



(a) t_{total}



(b) n_R

Figure 8: Experimental results for the real web-accessible data sets relevant to our New York City restaurant scenario.

7 Variations to our Model and Future Work

In this section we present some simple variations to our data and query models of Section 3, and some interesting directions we plan to investigate in the future.

Boolean (Filter) Conditions: Top-k queries can naturally incorporate Boolean (or filter) conditions over some attributes. Our algorithms can adapt to such conditions in a straightforward way if the conditions are over an *S-Source*: we can discard tuples, if appropriate, as soon as they are retrieved (or even in some cases the *S-Source*s might process the filter directly, as does the Yellow Pages source of Section 6). However, our algorithms need to be slightly modified to handle conditions over *R-Source*s. Specifically, we should probe the *R-Source*s for a filter predicate *before* probing the other *R-Source*s, to avoid "wasting" probes on objects that might be later discarded by the filter. If there are several filter predicates, we can probe them in *rank* order [11, 12].

Dealing with Real Web *R-Source*s**:** As mentioned above, some web sources might not fit our model perfectly. For instance, an *R-Source* R might not have information about some object returned by the *S-Source*. In such cases, we approximate the missing score with an expected value, which can be determined in a number of ways, including for example as an average of past scores observed from that source. A similar approach can be followed to handle sources that are down at query-processing time.

S-Source$s$ **that Return Several Objects at a Time:** Some sources (e.g., web search engines) return $n > 1$ matches at a time for a given query. Throughout our discussion, we assumed that *S-Source*s returned one object at a time. For our experiments with real web sources, we approximated the cost to retrieve each object by dividing the cost to retrieve a batch of n objects by n. However, there might be opportunities to refine the current algorithms by exploiting the fact that we can "look ahead" in the *S-Source* ranking at no extra cost.

Relaxing the Source Model: In Section 4 we focused on the case where exactly one *S-Source* and multiple *R-Source*s were available. In the general case, with more than one *S-Source,* we can divide the evaluation in two (pipelined) phases. First, we consider all *S-Source*s at a time and use the NRA algorithm [7] (which is instance optimal) to produce a stream of tuples sorted by the partial score of the *S-Source*s. Then, we treat this output as a new single *S-Source* and use the *Upper* algorithm of Section 4 to produce the final answer. Another direction is to combine these two phases into a single, more efficient algorithm. We are currently evaluating different alternatives for the latter approach.

Other Issues: Our model assumes that only one source can be accessed at a time, which is is too restrictive in the context of web sources. As explained in Section 2, we can incorporate the ideas in [9] to include parallelism and speed up the process. Another interesting problem is how to adapt our algorithms so that they take into consideration *S-Source*s that return just the rank of objects without the actual associated scores.

8 Conclusion

We studied techniques to efficiently evaluate top-k queries over web-accessible autonomous databases with a variety of access interfaces. In particular, we focused on web sources that can only be accessed via random accesses. We proposed extensions to existing algorithms for top-k queries so that they can handle random-access sources, and also introduced two novel strategies, *Upper* and *Pick*, which are designed specifically for our query model. We conducted a thorough experimental evaluation of these techniques using both synthetic and real web-accessible data sets. Our evaluation showed that *Upper* produces the best processing plans in terms of execution time for a variety of data and query parameters, and for both synthetic and real data sets.

References

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

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

[3] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, pages 91–102, 1996.

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

[5] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, pages 411–422, 1999.

[6] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems*, pages 216–226, 1996.

[7] R. Fagin, A. Lotem, and M. Laor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems*, 2001.

[8] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, 1999.

[9] R. Goldman and J. Widom. 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)*, pages 285–296, 2000.

[10] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases (VLDB'00)*, pages 419–428, 2000.

[11] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM International Conference on Management of Data (SIGMOD'93)*, pages 267–276, 1993.

[12] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD'94)*, pages 336–347, 1994.

[13] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases (VLDB'96)*, 1996.

[14] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 22–29, 1999.

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

[16] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, 1995.

[17] 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.