Guiding Queries to Information Sources with InfoBeacons

Brian F. Cooper

Center for Experimental Research in Computer Systems
College of Computing
Georgia Institute of Technology
cooperb@cc.gatech.edu

Abstract. The Internet provides a wealth of useful information in a vast number of dynamic information sources, but it is difficult to determine which sources are useful for a given query. Most existing techniques either require explicit source cooperation (for example, by exporting data summaries), or build a relatively static source characterization (for example, by assigning a topic to the source). We present a system, called *InfoBeacons*, that takes a different approach: data and sources are left "as is," and a peer-to-peer network of *beacons* uses past query results to "guide" queries to sources, who do the actual query processing. This approach has several advantages, including requiring minimal changes to sources, tolerance of dynamism and heterogeneity, and the ability to scale to large numbers of sources. We present the architecture of the system, and discuss the advantages of our design. We then focus on how a beacon can choose good sources for a query despite the loose coupling of beacons to sources. Beacons cache responses to previous queries and adapt the cache to changes at the source. The cache is then used to select good sources for future queries. We discuss results from a detailed experimental study using our beacon prototype which demonstrates that our "loosely coupled" approach is effective; a beacon only has to contact sixty percent or less of the sources contacted by existing, tightly coupled approaches, while providing results of equivalent or better relevance to queries.

Keywords: source discovery, integration, peer-to-peer search

1 Introduction

There is an explosion of useful data available from dynamic web sources, such as "deep-web" data sources [7], web services, web logs and personal web servers [3]. One reason that this information is so useful is that it is being constantly maintained and updated by a huge number of humans and software programs. The Internet and web standards make it possible and easy to contact a source and retrieve information. But the proliferation of sources creates a challenge: how to find the right source of information at a given point in time? Search engines are a useful tool for searching the "surface web" but most deep web data is not reachable via crawling, and much of it changes more quickly than search engines can keep up with.

A system which allows users to find the right information sources must deal simultaneously with four challenges.

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 59–78, 2004.

- *Scale* - there are millions of potential sources and a huge amount of aggregate data
- *Dynamism* - new sources are appearing and old sources are disappearing frequently, and the information in many sources is being updated constantly
- *Heterogeneity* - information structure and semantics vary widely between sites
- *Limited source cooperation* - while sources are willing to provide basic search and retrieval via HTML forms or a web services interface, they are frequently unwilling to export all of their data, change their query model, run foreign software, or otherwise modify their functionality

Existing source discovery systems deal with some of these challenges, but the most utility comes from addressing all of them.

We are developing a peer-to-peer middleware system called *InfoBeacons* to guide users to information sources while dealing with these four challenges. The functionality of InfoBeacons is based on that of a search engine over static web pages: a user submits a query to a *beacon*, and the beacon returns results. The user can then use these results, and may go directly to the information sources and perform more complex queries or browsing. If the beacon is unable to provide enough results, it routes the queries to neighbor beacons in the peer-to-peer overlay. Our InfoBeacons prototype operates on keyword queries, a "lowest common denominator" approach that works across a variety of sources, including text [1], XML [26] and relational data [21] and is intuitive to users.

The InfoBeacons system is designed around two basic principles. First, the system is composed of a set of *beacons*, lightweight middleware components that are loosely-coupled to several information sources. The loose coupling means that the beacons use the source's existing search and retrieval interface, without attempting complex semantic integration or requiring extra functionality on the part of the sources. This principle allows InfoBeacons to make the best of the limited cooperation from heterogeneous sources. Second, the system pushes most of the query processing to the sources themselves, while the beacons act mainly to choose sources and retrieve results. This principle ensures that the system scales to many sources, by utilizing the aggregate resources of the sources themselves and minimizing the load on each beacon. Also, processing queries at the sources ensures that the most current information is available to users.

One distinguishing feature of our system compared to other peer-to-peer systems is that the beacon cannot expect information sources to cooperate by exporting content summaries or notifying the beacon of changes. How can we choose good sources for queries in this situation? It is too expensive to broadcast the user query to all of the sources. Our approach is that beacons remember the results of previous user queries, and use these results to guide future queries. Unlike previous caching schemes (such as [27]), the InfoBeacons cache is not used to answer queries but instead to direct queries to the sources themselves. We introduce a function, called *ProbResults*, that ranks sources for a given query based on past results stored in the beacon's cache. We have also developed a heuristic, called *experience weighting*, that dynamically adapts the beacon cache based on the changing results returned by sources. Experiments with our InfoBeacons prototype on data gathered from the World Wide Web shows that a beacon using *ProbResults* and experience weighting can find high-quality information, without having to query a large number of sources and despite having limited infor-

mation. For example, a beacon using *ProbResults* contacts less than one quarter of the sources compared to a beacon using a random ordering, and only sixty percent of the sources compared to a beacon using a more tightly-coupled approach, even if sources are frequently changing their information. This results in less load on sources, as well as more than a factor of two decrease in query response time.

This paper presents the InfoBeacons architecture and explores our design choices. We then focus on one challenge faced by our architecture: how can a beacon choose good sources for a query despite limited cooperation from those sources? Since the beacon is the basic unit of functionality in our system, we must address this challenge before we can explore other aspects of the system. In ongoing work, we are addressing other issues, such as techniques for using multiple beacons to answer a query, and preliminary results are discussed in [13].

1.1 Related Work

Several peer-to-peer systems have been developed to provide information discovery, including multimedia filesharing systems (such as Kazaa and Gnutella), "unstructured" networks [11, 25] and distributed hash tables [31]. Each of these systems assumes the active participation of information sources to export content summaries to aid in source selection and query routing. However, many sources are unwilling to export their data, even those that provide free searching over their information. These sources may not want to expend the bandwidth necessary, may wish to protect their intellectual property by only serving individual results and not the whole collection, or may simply be unwilling to modify their existing server infrastructure. Our approach deals with sites that offer such limited cooperation. Also, our approach goes beyond previous P2P routing strategies that leverage result information (such as [25]) since we use whole document contents to achieve higher accuracy.

Similarly, several peer-to-peer systems have been developed to provide source integration [20, 22, 29]. Such systems can provide high retrieval accuracy but require either complex schema mappings between sites [4] or assume that all data is structured similarly [22]. In a large scale system such as the Web, it is too expensive to construct all the required mappings, and data is structured in a wide variety of ways.

The "source discovery" problem has been examined by a variety of investigators, including those in the fields of information retrieval [8], databases [17] and distributed systems [33]. Again, the dominant approach is to ask the source to export all of its data to a central broker, as in GlOSS [18] or CORI [16], or at least to export a summary of its data, as in YouSearch [3] or Galanis et al [17]. Such tight coupling works only if sources are willing to export data, and many are not. An alternate approach is to use query probes to build source content summaries or classify uncooperative sources [9, 19]. However, in a highly dynamic network, with lots of sources appearing and disappearing, and sources constantly updating their information, it may be difficult and expensive to keep these classifications accurate. In Section 5 we present experimental results comparing this approach to our techniques. Some systems assume a consistent classification scheme or topic hierarchy to which sources can be assigned (such as in [23, 32]), but it is not clear that sources can always be assigned a single, unambiguous topic or that a single hierarchy is equally useful to all users. Some systems combine

these two approaches, such as BrightPlanet [7]. The Harvest system was an early pioneer in database selection, with "brokers" similar to our beacons [6]. Harvest combined source data export with search engine-style crawling of static content through modules called "gatherers." Other systems focus on the mechanics of extracting structured data once a source has been found; an example is DeLa [34]. Such an extraction and transformation component could be added to the InfoBeacons system, which currently returns data to users in its raw form.

Caching of data to improve performance has been well studied in many contexts, including the web [2], database systems [15] information retrieval [27] and peer-to-peer search [5]. The most common use of caching in these systems is to cache data from a known source to hide latency. Search engines, such as Google or BrightPlanet, can be thought of as web caches that have the same goal as ours: directing users to sources that the users did not previously know about.

There are other types of systems that search across multiple sources, including data integration systems [10] and search engines [30]. Again, we are designing the InfoBeacons system to be more loosely coupled, so that tight data integration or centralized search engine summaries are not required. While the query semantics in InfoBeacons is consequently weaker than in a federated database, the system is more able to scale to very large numbers of sources since it avoides expensive schema and data integration. Moreover, search engines have difficulty dealing with frequently changing sources and data stored in "deep-web" databases, and our InfoBeacons architecture aims to address these challenges.

1.2 Contributions and Overview

In this paper, we examine InfoBeacons and show that a beacon can make good decisions about which sources to contact, without being tightly coupled to sources or building an *a priori* source classification. In particular, we make the following contributions:

- We describe the InfoBeacons architecture and our implemented prototype, and describe its benefits for finding web information sources (Section 2).
- We present *ProbResults*, a function for ranking sources based on previous results cached by the beacon (Section 3).
- We present *experience weighting*, a heuristic for online adaptation of the beacon cache to deal with changes at sources (Section 4).
- We report the results of a detailed experimental study that examines the performance of our beacon prototype on real web data. These experiments demonstrate that a beacons system, despite being loosely coupled to sources and having partial information, can find high quality information sources at low cost (Section 5).

Finally, in Section 6, we discuss conclusions and future work.

2 InfoBeacons Architecture

The InfoBeacons architecture is comprised of three basic elements: autonomous information sources, users and a peer-to-peer network of *beacons*. This architecture is shown in Figure 1.

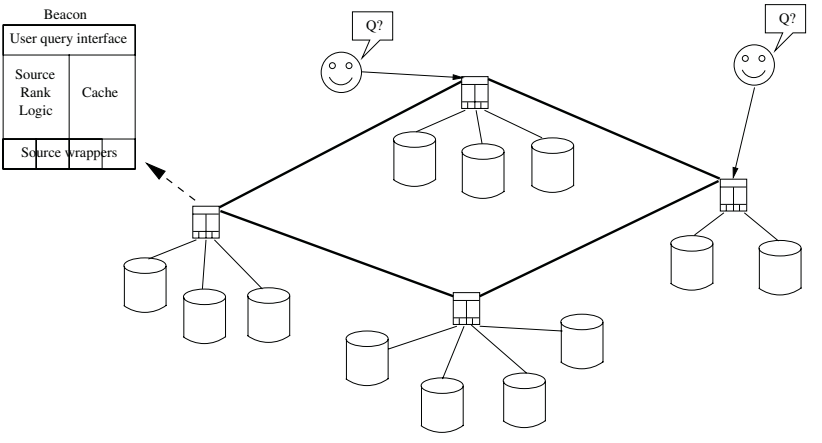


Fig. 1. InfoBeacons architecture.

Information sources provide an interface for searching and retrieving information. Examples of sources include databases searchable via HTML forms or web services that provide a SOAP interface. Each source has its own interface and query language, although we assume that they provide at least basic keyword search. Each source also has its own locally-defined semantics for keyword searching. For example, some sources may return documents matching all of the keywords, while others may return documents that are most similar to the query (even if some query words are missing). We envision that the InfoBeacons network should scale to at least thousands of sources and eventually to millions. We assume that sources process searches and return results free of charge, as many web sources do. The *documents* returned by sources may be of a variety of types, including HTML files, XML files, PDF files, relational tuples, and so on. Our current prototype deals with documents encoded in ASCII, although filters can be added to deal with other document types (such as PDF).

Users pose keyword queries in order to locate information sources. Users in our system are very similar to users of a traditional web search engine. For example, if they do not like the returned results, they will refine and resubmit their query. On the other hand, when a user gets a result that he likes, he may simply retrieve the matching document, or may go directly to the result source to browse and/or search more directly.

Users are directed to sources by *beacons*. Beacons have several responsibilities:

- Maintaining connection information for multiple sources
- Providing a uniform search interface to the user for all of the beacon's sources
- Submitting the user's queries to appropriate sources and returning results to the user
- Caching query results to aid in selecting appropriate sources for future queries
- Submitting the user's query to other beacons if the local beacon's sources provide inadequate results

As shown in Figure 1, each beacon contains several components in order to fulfill these responsibilities: a *user query interface* that allows the user to submit keyword queries,

a *cache* of query results, *source rank logic* to rank the desirability of sites for a given query based on the information on the cache, and *source wrappers* to send queries to sources and retrieve results. The site rank logic and cache are the subject of Sections 3 and 4. Techniques for generation of source wrappers have been studied by others; see for example [34]. For now, our beacon implementation merges results from multiple sources by ranking results by the local score returned by the source. More complex merging techniques (such as those in [14]) may be more useful although we have not yet investigated them.

Our goal is for beacons to be as lightweight as possible, so that they can run on a commodity machine (such as a PC). To achieve this goal, each beacon is only responsible for some of the sources, and cooperates with other beacons on other machines to share the user load. In general, beacons can be run by users, data sources, libraries, ISPs, and so on, and connect to each other in a peer-to-peer system. Techniques for beacon cooperation are examined in more detail in [13].

Note that beacons are loosely-coupled to sources. That is, a beacon needs only to know how to submit keyword queries to sources and retrieve results. The beacon does not need other information, such as the database schema or query processing paradigm, in order to guide users to sources. This allows beacons to manage multiple, heterogeneous sources without extensive data integration, and allows a beacon to add and remove sources from its list with minimal effort. There must be a mechanism for beacons to discover new sources. For example, the beacon can probe UDDI registries or search engines to find web sources. Alternatively, a specialized service can be developed to discover sources and assign them to beacons.

Consider an example system with multiple beacons $B_1, B_2 ... B_n$, each of which is responsible for several sources. A user might submit a query "houses for sale" to beacon B_3. Beacon B_3 may be responsible for sources $S_1, S_2 ... S_m$, and must choose an order in which to contact these sources. The beacon may decide to first submit the query "houses for sale" to S_5. Source S_5 may not return any results, and so the beacon next sends the query to S_{19}. Source S_{19} may return 15 results, which the beacon returns to the user. If the user wants more results, then the beacon will go to the next source on its list, perhaps S_{11}, and submit the query again. This process continues until the user has gotten enough results. The number of desired results may be specified when the query is first sent to the beacon, or may be determined interactively, with the user requesting "more results" until he is satisfied. If the beacon B_3 cannot find enough results, it would forward the query to other beacons, say, B_1 and B_9, to find more results. Our techniques must balance efficiency with the need to retrieve "good" results. As with a search engine, a beacon may not return the "best" results in the system, but as long as the results are good the user will be satisfied. In our experimental results (Section 5) we see that the quality of the beacon's results is in fact quite high.

The beacon must decide for each query which sources are most appropriate. To do this, the beacon sorts its sources in decreasing order of "desirability" on a per query basis, and then contacts the most desirable sources in order until enough results are found. The beacon, which both submits queries to the source and retrieves results for the user, can cache these results to aid source selection. If an information source returns only a URL (and possibly an abstract) for each document, the beacon can use that

URL to retrieve and cache the actual document. The issue of a good ranking function is examined in Section 3, where we propose several alternative ways to use the result cache to rank sources. If the beacon's sources do not provide enough information, the beacon must forward the query to other beacons until enough information is found. There are several possible ways to choose remote beacons for a query, and we are examining this issue in more detail in ongoing work [13].

2.1 Architecture Rationale

The InfoBeacons architecture provides a number of key advantages that allows it to scale to large numbers of information sources despite the limited cooperation from sources and a highly dynamic information environment.

First, InfoBeacons minimizes the requirements on sources. Sources continue to process queries (as they were already doing) but do not have to run extra foreign software to participate in the peer-to-peer system. Sources are not required to export their data to the peer-to-peer system for indexing, which is something many sources are unwilling to do for bandwidth or intellectual property reasons. Moreover, by intelligently selecting sources, InfoBeacons avoids the load on sources associated with sending them lots of irrelevant queries.

Second, new sources can be integrated into the system and searched by users with a minimum of effort. By avoiding complex semantic integration and focusing on keyword search, InfoBeacons reduces the problem of integrating a new source to the task of connecting to the source, submitting keyword queries and retrieving results. As with a search engine that intelligently selects sources, our middleware can provide high quality results without understanding the data semantics.

Third, the system can scale to large numbers of sources simply by adding new beacons. Each beacon is only responsible for some of the sources in the system, and thus the processing and storage requirements for a beacon is limited. Also, most of the processing is done by the information sources, as they process and answer keyword searches, further reducing the load on the beacons themselves. Therefore, it is feasible to deploy lots of lightweight beacons on commodity hardware scattered throughout the web.

Finally, beacons hide the complexity of connecting to and searching multiple information sources for most users. Only users that want to dig deeper into a particular source need to contact that source directly and search it themselves.

3 Choosing Data Sources

Beacons must intelligently choose which information sources will be sent user queries. The simplest approach, which is to send the query to all sources, is too expensive, both for the sources and for the beacons. Another simple approach is to contact the sources in random order until the user has received enough results. The random approach may reduce the cost, but beacons may potentially contact many sources that do not return results. Our approach is for the beacon to rank the sources for each query based on the likelihood that the source will return results for the query, and then contact the sources in that order. Previous results are used to estimate the usefulness of a given source for the current query.

Although the beacon cache could retain whole documents, doing so may require a large amount of memory, and our goal is to minimize the resource requirements of the beacon. For this reason, the cache consists only of a set of statistics about the result data. Specifically, for each source s, the beacon cache consists of a set of pairs $((W_1, CW_1^s), (W_2, CW_2^s), ...)$, where W_i is a word and CW_i^s is the count of W_i for source s. The exact meaning of CW_i^s is tied to the definition of *ProbResults* and is described below. To update the cache with a new document, the beacon must parse the document and extract its terms. Because the beacon cache stores only aggregated counts, and not whole documents, the cache is very compact. Experimental results in Section 5 show that high selection accuracy can be achieved with a cache that is only a few tens of megabytes. We may want to place an upper limit on the size of the cache, and then it is necessary to eject some of the (W_i, CW_i^s) pairs to save space if the cache becomes too large. We examine this possibility in Section 5.5.

In this section, we examine *ProbResults*, a technique that we have developed for using the beacon cache to rank sources. Our goal is to minimize the number of information sources contacted, while still providing useful results to users despite the incompleteness of the cached data. We also compare qualitatively to existing techniques for source selection. In Section 5 we present experimental results over real web data to compare techniques quantitatively.

3.1 ProbResults

The *ProbResults* ranking is based on the probability that an information source will return documents containing the query words if it is sent a query Q. We call this probability the *ProbResults score* for the site for a given query. The beacon will rank the information sources in order of decreasing *ProbResults* score. Sites with the same *ProbResults* score should be chosen in random order so that all sites that appear equally good have a chance of being chosen.

More formally, a query Q is a set of n_Q keywords $(QW_1, QW_2, ...QW_{n_Q})$. Different queries may have different numbers of keywords. Consider a source s that we have previously sent k_s queries too. Each query may contain different words, and may differ from Q. For each of these queries, s has returned zero or more results. The number of results that contain word W_i (whether or not it was part of the query) is stored in the cache as CW_i^s. The expected number of results from s containing query word QW_i is $PW_i^s = CW_i^s/k_s$.

The *ProbResults* score is the product of the PW_i^s values of the query words:

$$\textit{ProbResultsScore}_Q^s = \prod_{i=1}^{n_Q} PW_i^s = \prod_{i=1}^{n_Q} CW_i^s/k_s$$

Taking the product gives higher weight to sources that will return results relevant to all of the query words than to sources that are particularly relevant to one or two query words but not to the others. As a result, taking the product resulted in better experimental performance than the other ways we tried to combine the PW_i^s values into one score, including sum, max and min.

A beacon using *ProbResults* depends on cached result data to choose sources, and this may bias the beacon toward sources that have returned many results in the past at the expense of new sources or sources with less content. It may be possible to mitigate this bias by giving "extra credit" to new sources, by probabilistically choosing a low score source over a high score source, or by proactively probing new sources. We are examining these possibilities in ongoing work.

Minimum Probability. The beacon cache contains incomplete information about a source; in particular, it may not contain all of a source's documents. As a result, PW_i^s may be zero for a word that actually does appear in documents at a source. Because $\mathit{ProbResultsScore}_Q^s$ is the product of PW_i^s values for a source, the effect is that a source may be given a *ProbResults* score of zero and placed at the bottom of the ranking if there is no cache information for one or more query words for that source.

To see why this is a problem, consider a query for words "exothermic reactions." A beacon may have cached documents from source s_1 containing the word "exothermic," but no documents containing "reactions." The same beacon may have a second source s_2, but no cached documents containing either "exothermic" or "reactions" for s_2. In this case, s_1 should clearly be queried before s_2, since s_1 is more likely to have documents containing the query words. However, the *ProbResults* score for both sources will be zero, as neither contains cached information for "reactions."

We address this problem by using a special constant PW^{min}, $0 < PW^{min} \leq 1$, instead of zero, for the PW_i^s probability when we have no cached information for word QW_i for source s. Formally:

$$PW_i^s = \begin{cases} CW_i^s/k_s \text{ if } CW_i > 0 \\ PW^{min} \quad \text{otherwise} \end{cases}$$

Our experience in building the InfoBeacons prototype has taught us that choosing a good value for PW^{min} can have a big impact on system performance. In Section 5.2 we investigate appropriate values for PW^{min} using experiments. Alternatively, a good value of PW^{min} can be learned adaptively by each beacon.

Note that other systems have also dealt with incomplete content summaries; see for example [23], which uses a hierarchy of summaries to infer missing information. Our PW^{min} technique is simpler than the techniques used in many of these systems, but still produces good performance in practice.

Existing Techniques. The *ProbResults* ranking function is similar to the *Ind* ranking function used in bGlOSS [18]. *Ind* ranks sources based on the content at sources, while *ProbResults* ranks sources based on the behavior of those sources in response to queries. In other words, if a source returns document D_1 100 times and document D_2 once, *Ind* treats both documents as equally descriptive of a source, while *ProbResults* would place more weight on the words in D_1. This distinction allows *ProbResults* to better predict what a source's response to a query will be. *Ind* was developed to work with conjunctive boolean query sources; that is, sources that only return documents containing all of the query words. Our *ProbResults* function attempts only to characterize the results

returned for queries, not the source's query model, and thus works well across a variety of sources, including boolean and vector space model sources (e.g. sources that return documents based on "similarity" to the query rather than exact match of all query words).

Another version of GlOSS, vGlOSS, was designed to work directly with vector space model sources. The vGlOSS system uses the *Max* metric to rank sources. *Max* attempts to predict the scores a source will give different documents for a given query, and then uses these scores to predict the number of documents that will have a score greater than some user-defined threshold l. As a result, vGlOSS must understand the source's query model, and in particular, the source must send a list of documents, words and word frequencies to vGlOSS. The full definition of *Max* given in [18].

In *CORI* [16], each database is ranked as a function of two statistics: df_i^s, the number of documents at source s containing QW_i, and sf_i, the *source frequency* (number of sources that have documents containing QW_i). A source gets a high score for a query if the query contains words that appear frequently in the source and infrequently in other sources. The formula for *CORI* is given in [16].

Ind, *Max* and *CORI* require the source to export all of its data to a central index before any queries are processed. If a source refuses to export its data, Callan and Connell [9] have suggested building a source characterization by sending a set of randomly selected words, or *query probes*, to collect a subset of the source's documents (again, before any queries are processed). Then, *Ind*, *Max* or *CORI* can be used to select sources based on these collected documents.

In contrast, in our approach the beacon cache is continually updated as new results arrive. Therefore, the information the beacon has about each source continually improves as time progresses. Moreover, the beacon cache is updated with data that matches the queries users are actually asking for, so that the cached data is focused on information that users are interested in. In pre-caching, all documents are cached, and in query probing, the set of cached documents depends on randomly chosen query probes. By utilizing the results of past queries to load the cache, beacons can form accurate source characterizations even without explicit source cooperation.

In Section 5 we present experimental results comparing our *ProbResults* ranking and caching techniques against *Ind*, *Max* and *CORI* with both pre-caching and query probing.

4 Cache Forgetting and Experience Weighting

When a beacon cache contains a word for a given source, the beacon has some reason to believe that future queries containing the word will return results from that source. However, experience may prove otherwise for two reasons. First, the data at sources may be changing frequently, and thus the cached information may have become out of date. Second, just because a source returns a document containing a given word does not necessarily mean that the word is a useful query term for that source. For example, a source of weather information may have the word "weather" in all of its documents. However, many searching techniques (such as TF/IDF weighting in information retrieval [1]) give a query term very low weight if it is too common, and a query for "weather" will not produce any results from that source.

A common solution is to use a *forgetting factor* μ to decrease the importance of old samples relative to new samples; this technique is used in areas such as control systems and reinforcement learning [24]. In our context, we would periodically multiply each CW_i^s value in the cache by μ. (Recall that the beacon cache maintains for each source s a set of (word,count) pairs $((W_1, CW_1^s), (W_2, CW_2^s), ...)$.) The effect is that truly stale information (such as that representing documents removed from the system) would slowly be "forgotten," while up to date information (such as that representing new documents or old documents that are still in the system) would be refreshed by the results of new queries. One potential disadvantage of a forgetting factor is that even old information can be useful in characterizing a source, especially if there is little new information. Another potential disadvantage is that the forgetting factor deals primarily with the issue of stale cache data and does not directly address the second issue listed above, that it is difficult to predict the weight a source will assign to a term.

We have developed another technique that allows the cache to dynamically adapt to align with the results that sources are actually producing. We call this heuristic *experience weighting*. The basic idea is that we weight the word counts in the cache based on the beacon's experience with each word as a query term. Then, when a beacon receives a query with a given word, the beacon is more likely to send the query to a source that has produced results before for queries with that term than sources that have been queried with that term and produced no results.

Experience weighting operates as follows. We specify an *experience factor* $EF \geq 1$. After each query, the beacon updates the cache count CW_i^s of each query word QW_i for each contacted source s:

- If the source returned results, CW_i^s is multiplied by EF
- If the source returned zero results, CW_i^s is divided by EF

Note that $EF = 1$ is equivalent to no experience weighting.

After applying the experience weighting heuristic to the cache, the *ProbResults* score no longer has a strict probability interpretation. However, the general intuition behind each function still applies: sources are given a higher score if they are more likely to provide good result for a query.

The magnitude of the impact on the cache from forgetting and experience weighting depends on the value of the forgetting factor μ and the experience factor EF. Appropriate values can be found using experiments, and in Section 5.2 we examine results that identify good values these parameters, as well as examine the effectiveness of forgetting versus experience weighting. Alternatively, EF or μ can be set adaptively, based on the experience of the beacon.

5 Experimental Results

We have run a set of experiments to evaluate our architecture and techniques. We examine two types of metrics: cost and quality. Our *cost* metric is the number of sources that are contacted for each query. Contacting fewer sources is better because the load on sources and beacons is reduced, and response time and throughput are improved.

To measure the quality of results returned by our system, we use the cosine distance with *TF/IDF* weighting, a common measure of relevance in information retrieval sys-

Table 1. Data and workload values.

Number of documents	169,902
Total data size	4.04 GB
Information sources	100
Documents per information source	111...5,517
Queries	100,000
N_Q: Terms per query	1...6
DT: Document turnover	0%, 50%
T: User result threshold	10

tems [1]. In this metric, both documents and queries are represented as term vectors, and the relevance of a query to a document is calculated as the cosine distance between the two vectors (e.g., the inner product). Term vectors are weighted based on the *inverse document frequency* (IDF); terms that appear less frequently are more descriptive and are given higher weights. We calculated IDF over all documents in the system. Thus, the TF/IDF score for a document returned for a query represents how relevant that document is compared to all documents at all sources. The "quality" of a query's results is the total TF/IDF score of documents found by the beacon for the query.

In the following sections, we examine the performance of using a beacon to choose information sources. In summary, our results show:

- The *ProbResults* ranking has lowest cost, especially in scenarios where there are frequent updates of source data (contacting 40 to 45 percent fewer sources than *CORI*, *Ind* or *Max*).
- The *ProbResults* ranking also find the highest quality documents, with an average total TF/IDF score of 3.1 versus 3.0 or less for *CORI*, *Ind* or *Max*.
- Our prototype took 4.3 seconds per query on average to select sources for queries, query those sources over the Internet, and retrieve, cache and return results to the user. In comparison, a beacon choosing sources randomly required 11.8 seconds per query.
- The cache of the beacon is quite compact, requiring only a few tens of megabytes in our experiments. If we limit the cache size, the performance degrades gracefully.
- Our conclusions hold even as we change the number of sources.

5.1 Experimental Setup

The characteristics of our data and workload are shown in Table 1. We discuss these values in this section.

Our InfoBeacons prototype is written in C++, and currently runs on Unix and Linux platforms. A beacon accepts user queries and returns results via XML over HTTP.

To ensure our experiments were repeatable, we created our own information sources on machines in our lab, and populated them with HTML documents downloaded from 100 .com, .net, .gov, .edu and .org websites. (In the extended version of this paper [12], we discuss results for searching across larger numbers of sites and sites with different numbers of documents.) Each information source managed documents downloaded

from one website, and processed keyword searches using TF/IDF weighting and the inner product of the document and query vectors. We set each source to return only "relevant" documents, that is, those that had a score of at least 0.1 (on a scale from 0 to 1). The beacon contacted each source using XML over HTTP. Some sources had many documents and some had few, just as we would expect to find on the actual web.

We used synthetically generated keyword queries so that we could evaluate our system with a very large query set. For each query, we randomly selected N_Q words from a randomly selected document. Within our query set the N_Q value varied between one and six. The probability that a given word was chosen was a normally distributed function of the total number of occurrences of the term in all documents in the system. The mean μ of the normal distribution was equal to the average number of occurrences of any term in all documents, and the standard deviation was $\sigma = \mu/2$. As a result, the most frequent terms are terms that are neither too common nor too rare in the document corpus. This distribution of query terms matches the observed distribution of several real query sets as reported in [8].

We assume that each user has a threshold T: the number of desired document results. This is similar to a search engine, where users usually only look at the first page or two of results. Although we used $T = 10$ in the results presented here, other experiments (omitted here) show that our results and techniques generalize to other values of T. For example, the number of sources contacted under each ranking function increases with T, but for all values of T, *ProbResults* was the best ranking function in terms of cost, and the advantage of *ProbResults* over other functions increased as T increased. We also experimented with an alternate model, where a user wanted T sources instead of T results; this model is examined in [12].

We examined two scenarios: a *static* scenario and a *dynamic* scenario. In the static scenario, the documents at sources do not change. In the dynamic scenario, some documents are added and removed during querying. The document turnover was 50 percent; that is, 50 percent of the total documents were added and 50 percent of the total documents were removed while queries were being processed. The dynamic scenario models sources that are changing frequently, adding and removing information.

5.2 Tuning *ProbResults* with PW^{min} and Experience Weighting

PW^{min}. First, we examine the effect of using the minimum probability PW^{min} on the beacon's performance. Recall that the PW^{min} value determines the score assigned to a word for a source if that source has no information cached about the word. Figure 2 shows the effect of PW^{min} on the performance of a beacon using *ProbResults* in the static scenario with $EF = 1$. As the figure shows, choosing the right PW^{min} value can have a large effect on performance. The best value, $PW^{min} = 0.0001$, results in only 10.0 sites contacted on average. This represents a 64 percent decrease over $PW^{min} = 0$ (27.8 sites contacted) and an 86 percent decrease over $PW^{min} = 1$ (70.0 sites contacted). In other words, carefully selecting a PW^{min} that is non-zero and less than one is key to achieving good performace. Similar results (not shown) were obtained for the dynamic scenario, and for $EF \neq 1$. These results also show that a carefully chosen PW^{min} is important, and that $PW^{min} = 0.0001$ works best.

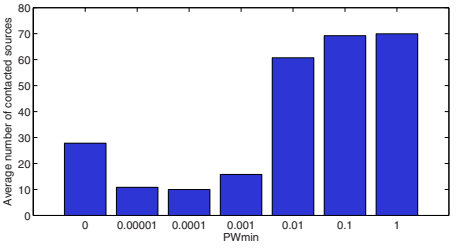
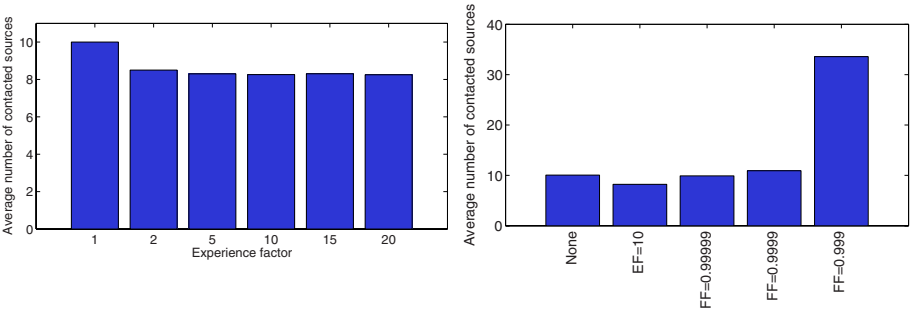


Fig. 2. Effect of PW^{min} on *ProbResults*.

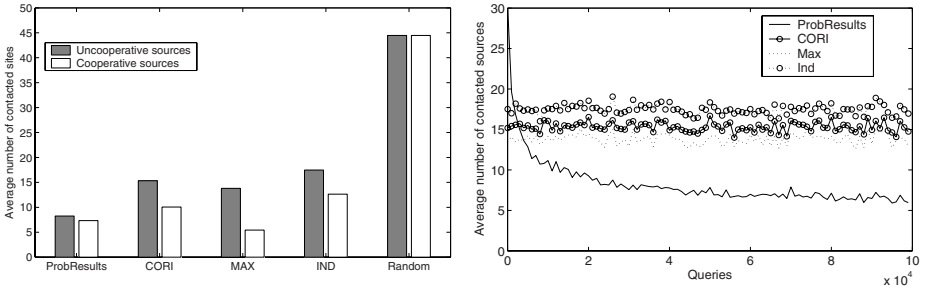


(a) (b)

Fig. 3. Cache adaptation with *ProbResults*: (a) Experience factor EF, (b) Forgetting factor FF.

Experience Weighting Versus Forgetting Factor. The experience factor is also key to performance. Recall that $EF = 1$ is equivalent to no experience weighting, while $EF \neq 1$ means weighting the cache to reflect the results that sources are returning. Figure 3(a) shows the effect of the experience factor on the performance of *ProbResults* in the static scenario with $PW^{min} = 0.0001$. As the figure shows, using experience weighting can have a significant effect: the best value, $EF = 10$, results in 8.3 sources contacted on average, 17 percent less than no experience weighting ($EF = 1$, 10.0 sources contacted). Note that increasing the experience factor beyond 10 does not provide any further improvement. Again, results are similar for the dynamic scenario.

We also compared the effectiveness of using experience weighting versus a forgetting factor. Figure 3(b) shows the results, with various forgetting factors (FF=X), $EF = 10$, and no cache adaptation at all (neither experience weighting nor forgetting; marked "None"). Again, $PW^{min} = 0.0001$. As the figure shows, the experience factor is more effective than the forgetting factor, with $EF = 10$ resulting in 8.3 sources contacted on average, 17 percent less than the best forgetting factor ($FF = 0.99999$, 9.9 sources contacted on average). In fact, for other forgetting factors, forgetting actually produces worse results than no cache adaptation at all. This is because forgetting can cause the cache to lose what little information it has about some sources, and thus the beacon begins to make bad decisions. Experience weighting retains all cache information, but weights the most useful information most heavily.



(a) (b)

Fig. 4. Comparison of ranking functions - static scenario: (a) overall average, (b) running average.

For the rest of our discussion, we will use $PW^{min} = 0.0001$ and $EF = 10$ for *ProbResults*.

5.3 Comparison – Static Scenario

Performance. We can now examine the performance of various ranking functions used by the beacon. In this section, we first look at the number of sources contacted, and second at the quality of results. We compare four ranking functions:

- *ProbResults*: our ranking function based on the probability that a source returns results
- *CORI*: the ranking function from the CORI system
- *Max*: the ranking function from the vGlOSS system
- *Ind*: the ranking function from the bGlOSS system
- *Random*: sources are selected in random order (similar to a random walk [28])

We found that the *Ind* ranking performed significantly better if we used a minimum probability $PW^{min} = 0.0001$ in the same way as in *ProbResults*. Like *ProbResults*, if a cache count for a query word is zero or missing, the source score will be zero. Using a PW^{min} value avoids ranking promising sources low simply because the cache is incomplete, resulting in a factor of two improvement in performance for *Ind*. Therefore, the result we report represent *Ind* modified to use a minimum probability.

The results for the static scenario, averaged over all 100,000 queries, are shown in Figure 4(a). Consider first the uncooperative source scenario (gray bars), which is our focus in this paper. In our experiment, *CORI*, *Max* and *Ind* use query probing to deal with uncooperative sources. As the figure shows, *ProbResults* performs better than the other functions. A beacon using *ProbResults* contacts 8.3 sources on average, compared to 15.3 for *CORI*, 13.8 for *Max* and 17.4 for *Ind*. Random is significantly worse. By accurately characterizing a source's behavior instead of just its content, *ProbResults* is best able to predict which sources will return results for a given query.

In fact, the performance of *ProbResults* continually improves as its cache becomes more accurate. Figure 4(b) shows a running average of the performance of each function

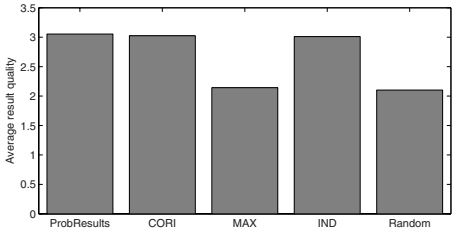


Fig. 5. Quality comparison of ranking functions - static scenario.

in the uncooperative sources scenario, with averages calculated every 1,000 queries. As the figure shows, initially *ProbResults* performs poorly, but after about 6,000 queries, *ProbResults* begins to perform better than all the other functions. After 100,000 queries, a beacon using *ProbResults* only has to contact 6.0 sources on average, less than half of the sources contacted using *CORI*, *Max* or *Ind*. Clearly, the combination of *ProbResults*, result caching and experience weighting is significantly better than previous methods.

Recall that *CORI*, *Max* and *Ind* were originally designed to operate on a full mirror of each source's content, which requires that the sources export all of their data. Although uncooperative sources are unlikely to export data, it is interesting to compare *ProbResults* to the performance of *CORI*, *Max* and *Ind* when using a full mirror. This "cooperative sources" scenario is also shown in Figure 4(a) (white bars). *ProbResults* with or without source cooperation outperforms *CORI* and *Ind*, even when those functions have source cooperation. This is because that characterizing the behavior of sources can be as important as, or more important than, characterizing their content, and *ProbResults* accurately characterizes source behavior. With cooperative sources, *Max* performs best overall, contacting only 5.4 sources on average compared to 7.3 for *ProbResults*. Thus, *ProbResults* is most appropriate for the uncooperative scenario.

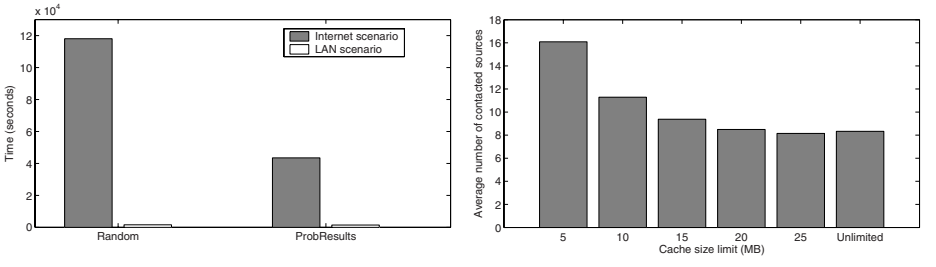
Quality. Next, we examine the quality of the results found by the beacon. Recall that we use TF/IDF weighting and cosine distance as our quality metric; this metric is a common measure of relevance in information retrieval. Figure 5 shows the results for the static scenario. As the figure shows, a beacon returns the highest quality documents with *ProbResults* (3.1), *CORI* (3.0) and *Ind* (3.0). *Max* and Random provide lower quality results (2.1 each). While *ProbResults* does not provide significantly more quality than *CORI* or *Ind*, it does provide high quality results when compared to these traditional functions. This result shows that the improved performance of *ProbResults* reported in the previous section does not cause a corresponding decrease in the quality of results.

5.4 Comparison – Dynamic Scenario

In the dynamic scenario, documents are added and removed from sources as the experiment progresses so we can measure the performance of the various techniques for dynamic sources. Due to space limitations, the full results are presented in the extended version of this paper [12]. In summary, the *ProbResults* performance continued to offer the best performance, contacting 45 percent fewer sources than *CORI*, *Max* or *Ind* while still providing the highest quality information.

Table 2. Machine characteristics.

	Internet	*LAN*
Beacon	Dell 4 x 2.0 GHz Xeon, 6 GB RAM	Dell 8 x 550 MHz Xeon, 4 GB RAM
Sources	HP RX2600 2 x 900 Mhz Itanium II 6 GB RAM	HP RX2600 2 x 900 Mhz Itanium II 6 GB RAM
Ping	69.9 ms	< 1 ms



(a) (b)

Fig. 6. Time and memory: (a) time to process 10,000 queries, (b) limiting cache size.

5.5 Time and Memory

Next, we ran a set of experiments to measure the time and memory requirements for our beacon prototype. To do this, we constructed two scenarios. In the *Internet* scenario, the beacon ran on a machine at Stanford University, while the data sources ran on a machine at Georgia Tech[1]. Thus, the beacon had to communicate with the sources using the Internet, incurring high latency for every roundtrip. In the *LAN* scenario, the data sources ran on the same machine at Georgia Tech, but now the beacon ran on another nearby Georgia Tech machine. In this scenario, the beacon contacted sources via gigabit Ethernet. In both scenarios, a client program, running on the same machine as the beacon, connected to the beacon via HTTP to submit queries and retrieve results. The characteristics of the machines involved are listed in Table 2. The average ping time between the beacon machine and the source machine was 69.9 ms in the Internet scenario, and less than 1 ms in the LAN case. We expect the Internet scenario to be most representative of a system of beacons querying real web sources.

In our experiment, we warmed the beacon's cache using the first 90,000 queries of our query set. (This was done using sources local to the beacon machine to save time in conducting the experiment.) Then, we measured the time required to process the next 10,000 queries of our query set.

The results are shown in Figure 6(a) for *ProbResults* ($MP = 0.0001$, $EF = 10$). The figure also shows the time for the beacon to process the same 10,000 queries using the Random ranking function. As the figure shows, in the Internet scenario, the beacon

[1] We would like to thank the Database Group at Stanford University for the use of their machine.

performs 2.7 times as fast using the *ProbResults* function (4.3 seconds per query) than when using the Random ranking (11.8 seconds per query). This difference shows how intelligently selecting sources can improve the response time of beacon queries, in addition to other benefits such as not overloading sources. The response time improvement of *ProbResults* versus Random is not quite as dramatic as the decrease in the number of sources contacted shown in Figure 4(a). While the number of sources contacted is less under *ProbResults*, the amount of data transferred once good sources are found is roughly constant under both methods, and this data transfer incurs large latency.

In the LAN scenario, the two methods perform almost equally: 0.144 seconds per query under *ProbResults* versus 0.158 seconds per query under Random. Because network latency is low, most of the performance improvement from contacting less sources is mitigated by the time the beacon takes to parse and cache the documents in the *ProbResults* method, a process that is not necessary in the Random case.

We also measured the memory requirements of the beacon. After 100,000 queries, the *ProbResults* cache required 64.6 MB of RAM: 28.7 MB for the document words and associated counts, and 35.9 MB for a set of hashtables, one per source, to index the words. These results demonstrate that even with a moderate cache size the beacon can make good decisions about which sources to contact.

Nonetheless, a user may wish to limit the size of the beacon cache. In our beacon implementation, a user can set a cache size limit. When the cache exceeds this limit, the beacon will eject words and associated counts (e.g., (W_i, CW_i^s) pairs) from the cache until the size is under the limit. The beacon ejects words in order of increasing $|CW_i^s - PW^{min}|$ (where $|A|$ is the absolute value of A) so that the counts are closest to PW^{min}, which have the least useful information, are ejected first.

The results are shown in Figure 6(b), where the horizontal axis shows the cache size limit (in terms of the size of the cached document words and associated counts). As the figure shows, the beacon's performance degrades gracefully as the cache becomes smaller. Even for a very small cache of 5 MB, the beacon using *ProbResults* contacts only about twice the sources of a beacon with unlimited cache, and has performance roughly equivalent to a beacon using *Ind*, *Max* or *CORI* (e.g., Figure 4). We can see from these results that the beacon is quite effective at choosing information sources, even with limited cache size.

5.6 Beacon Scalability

In most of our experiments, the beacon was responsible for 100 sources. We also examined results for different numbers of sources to see how the beacon scaled to larger source sets. Due to space limitations, these results are discussed in the extended version of this paper [12]. In summary, the performance of our techniques was not negatively impacted as we increased the number of sources.

6 Conclusions and Future Work

We have presented *InfoBeacons*, a system designed to process information from large numbers of diverse web information sources. The design philosophy behind our system is to loosely couple beacons to web sources, so that no modifications are needed

to the sources and so our beacon can adapt quickly to changes. This approach results in a number of benefits: many more sources can participate in the system, the system can scale well despite heterogeneity and dynamism, and the most up to date information can be located and retrieved. However, because beacons have limited information about sources, beacons must make the best of the information they have in order to select sources for queries. We presented *ProbResults*, a ranking function that uses cached information from previous queries to choose sources. We also described experience weighting, a heuristic that allows a beacon cache to adapt effectively to changes at sources. Experimental results show that a beacon using *ProbResults* and experience weighting can find high quality results while contacting forty to forty-five percent fewer sources than existing techniques. Our focus in this paper has been on the architecture and the source selection aspects of InfoBeacons. Another important aspect is the distributed cooperation of multiple beacons. The good performance of the *ProbResults* function suggests that it may be useful as the basis of a routing function that can choose beacons in a manner analogous to how beacons choose sources. We are examining this possibility in ongoing work [13]. Overall, our results show that the InfoBeacons framework is a promising middleware architecture for distributed information source discovery.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, N.Y., 1999.
2. G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, May 2000.
3. M. Bawa, R. J. Bayardo Jr., S. Rajagopalan, and E. Shekita. Make it fresh, make it quick — searching a network of personal webservers. In *Proc. WWW*, 2003.
4. P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *Proc. WebDB*, 2002.
5. B. Bhattacharjee. Efficient peer-to-peer searches using result-caching. In *Proc. IPTPS*, 2003.
6. C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. In *Proc. 2nd WWW Conference*, 1994.
7. BrightPlanet. Deep content. http://www.brightplanet.com/deepcontent/index.asp, 2003.
8. B. Cahoon, K. S. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, 18(1):1–43, January 2000.
9. J.P. Callan and M.E. Connell. Query-based sampling of text databases. *ACM TOIS*, 19(2):97–130, 2001.
10. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *In Proc. of IPSJ Conference*, October 1994.
11. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proc. ACM SIGCOMM*, 2003.
12. B.F. Cooper. Guiding users to information sources with InfoBeacons (extended version). Technical Report, at http://www.cc.gatech.edu/~cooperb/pubs/infobeaconsext.pdf, 2004.
13. B.F. Cooper. Using information retrieval techniques to route queries in an InfoBeacons network. Technical Report, at http://www.cc.gatech.edu/~cooperb/pubs/beaconrouting.pdf, 2004.

14. R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proc. SIGMOD*, 2003.
15. M.J. Franklin and M.J. Carey. Client-server caching revisited. In *Proc. Int'l Workshop on Distributed Object Management*, 1992.
16. J.C. French, A.L. Powell, J. Callan, C.L. Viles, T. Emmitt, K.J. Prey, and Y. Mou. Comparing the performance of database selection algorithms. In *Proc. SIGIR*, 1999.
17. L. Galanis, Y. Wang, S.R. Jeffrey, and D.J. DeWitt. Locating data sources in large distributed systems. In *Proc. VLDB*, 2003.
18. L. Gravano, H. Garcia-Molina, and A. Tomasic. GlOSS: Text-source discovery over the internet. *ACM TODS*, 24(2):229–264, June 1999.
19. L. Gravano, P.G. Ipeirotis, and M. Sahami. QProber: A system for automatic classification of hidden-web databases. *ACM TOIS*, 21(1):1–41, January 2003.
20. A.Y. Halevy, Z.G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proc. WWW*, 2003.
21. V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proc. VLDB*, 2003.
22. R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. VLDB*, 2003.
23. P. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *Proc. VLDB*, 2002.
24. L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, May 1996.
25. V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. CIKM*, 2002.
26. C. Botev L. Guo, F. Shao and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *Proc. SIGMOD*, 2003.
27. Z. Lu and K. S. McKinley. Partial collection replication versus caching for information retrieval systems. In *Proc. SIGIR*, 2000.
28. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of ACM Int'l Conf. on Supercomputing (ICS'02)*, June 2002.
29. W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Loser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proc. WWW*, 2003.
30. L. Page and S. Brin. The anatomy of a large-scale hypertext web search engine. In *Proc. WWW*, 1998.
31. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, 2003.
32. A. Sugiura and O. Etzioni. Query routing for web search engines: Architecture and experiments. In *Proc. WWW*, 2000.
33. C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. SIGCOMM*, 2003.
34. J. Wang and F. Lochovsky. Data extraction and label assignment for web databases. In *Proc. WWW*, 2003.