Chapter 8

A SURVEY OF ALGORITHMS FOR KEYWORD SEARCH ON GRAPH DATA

Haixun Wang
Microsoft Research Asia
Beijing, China 100190
haixunw@microsoft.com

Charu C. Aggarwal
IBM T. J. Watson Research Center
Hawthorne, NY 10532
charu@us.ibm.com

Abstract In this chapter, we survey methods that perform keyword search on graph data. Keyword search provides a simple but user-friendly interface to retrieve information from complicated data structures. Since many real life datasets are represented by trees and graphs, keyword search has become an attractive mechanism for data of a variety of types. In this survey, we discuss methods of keyword search on schema graphs, which are abstract representation for XML data and relational data, and methods of keyword search on schema-free graphs. In our discussion, we focus on three major challenges of keyword search on graphs. First, what is the semantics of keyword search on graphs, or, what qualifies as an answer to a keyword search; second, what constitutes a good answer, or, how to rank the answers; third, how to perform keyword search efficiently. We also discuss some unresolved challenges and propose some new research directions on this topic.

Keywords: Keyword Search, Information Retrieval, Graph Structured Data, Semi-Structured Data

1. Introduction

Keyword search is the *de facto* information retrieval mechanism for data on the World Wide Web. It also proves to be an effective mechanism for querying semi-structured and structured data, because of its user-friendly query interface. In this survey, we focus on keyword search problems for XML documents (semi-structured data), relational databases (structured data), and all kinds of schema-free graph data.

Recently, query processing over graph-structured data has attracted increasing attention, as myriads of applications are driven by and producing graph-structured data [14]. For example, in semantic web, two major W3C standards, RDF and OWL, conform to node-labeled and edge-labeled graph models. In bioinformatics, many well-known projects, e.g., BioCyc (http://biocyc.org), build graph-structured databases. In social network analysis, much interest centers around all kinds of personal interconnections. In other applications, raw data might not be graph-structured at the first glance, but there are many implicit connections among data items; restoring these connections often allows more effective and intuitive querying. For example, a number of projects [1, 18, 3, 26, 8] enable keyword search over relational databases. In personal information management (PIM) systems [10, 5], objects such as emails, documents, and photos are interwoven into a graph using manually or automatically established connections among them. The list of examples of graph-structured data goes on.

For data with relational and XML schema, specific query languages, such as SQL and XQuery, have been developed for information retrieval. In order to query such data, the user must master a complex query language and understand the underlying data schema. In relational databases, information about an object is often scattered in multiple tables due to normalization considerations, and in XML datasets, the schema are often complicated and embedded XML structures often create a lot of difficulty to express queries that are forced to traverse tree structures. Furthermore, many applications work on graph-structured data with no obvious, well-structured schema, so the option of information retrieval based on query languages is not applicable.

Both relational databases and XML databases can be viewed as graphs. Specifically, XML datasets can be regarded as graphs when IDREF/ID links are taken into consideration, and a relational database can be regarded as a data graph that has tuples and keywords as nodes. In the data graph, for example, two tuples are connected by an edge if they can be joined using a foreign key; a tuple and a keyword are connected if the tuple contains the keyword. Thus, traditional graph search algorithms, which extract features (e.g., paths [27], frequent-patterns [30], sequences [20]) from graph data, and convert queries into searches over feature spaces, can be used for such data.

However, traditional graph search methods usually focus more on the structure of the graph rather than the semantic content of the graph. In XML and relational data graphs, nodes contain keywords, and sometimes nodes and edges are labeled. The problem of keyword search requires us to determine a group of densely linked nodes in the graph, which may satisfy a particular keyword-based query. Thus, the keyword search problem makes use of *both* the content and the linkage structure. These two sources of information actually re-enforce each other, and improve the overall quality of the results. This makes keyword search a more preferred information retrieval method. Keyword search allows users to query the databases quickly, with no need to know the schema of the respective databases. In addition, keyword search can help discover unexpected answers that are often difficult to obtain via rigid-format SQL queries. It is for these reasons that keyword search over tree- and graph-structured data has attracted much attention [1, 18, 3, 6, 13, 16, 2, 28, 21, 26, 24, 8].

Keyword search over graph data presents many challenges. The first question we must answer is that, what constitutes an answer to a keyword. For information retrieval on the Web, answers are simply Web documents that contain the keywords. In our case, the entire dataset is considered as a single graph, so the algorithms must work on a finer granularity and decide what subgraphs are qualified as answers. Furthermore, since many subgraphs may satisfy a query, we must design ranking strategies to find top answers. The definition of answers and the design of their ranking strategies must satisfy users' intention. For example, several papers [16, 2, 12, 26] adopt IR-style answer-tree ranking strategies to enhance semantics of answers. Finally, a major challenge for keyword search over graph data is query efficiency, which to a large extent hinges on the semantics of the query and the ranking strategy. For instance, some ranking strategies score an answer by the sum of edge weights. In this case, finding the top-ranked answer is equivalent to the group Steiner tree problem [9], which is NP-hard. Thus, finding the exact top k answers is inherently difficult. To improve search efficiency, many systems, such as BANKS [3], propose ways to reduce the search space. As another example, BLINKS [14] avoids the inherent difficulty of the group Steiner tree problem by proposing an alternative scoring mechanism, which lowers complexity and enables effective indexing and pruning.

Before we delve into the details of various keyword search problems for graph data, we briefly summarize the scope of this survey chapter. We classify algorithms we survey into three categories based on the schema constraints in the underlying graph data.

- **Keyword Search on XML Data:**

  Keyword search on XML data [11, 6, 13, 23, 25] is a simpler problem than on schema-free graphs. They are basically constrained to tree structures, where each node only has a single incoming path. This property provides great optimization opportunities [28]. Connectivity information can also be efficiently encoded and indexed. For example, in XRank [13], the Dewey inverted list is used to index paths so that a keyword query can be evaluated without tree traversal.

- **Keyword Search over Relational Databases:**

  Keyword search on relational databases [1, 3, 18, 16, 26] has attracted much interest. Conceptually, a database is viewed as a labeled graph where tuples in different tables are treated as nodes connected via foreign-key relationships. Note that a graph constructed this way usually has a regular structure because schema restricts node connections. Different from the graph-search approach in BANKS [3], DBXplorer [1] and DISCOVER [18] construct join expressions and evaluate them, relying heavily on the database schema and query processing techniques in RDBMS.

- **Keyword Search on Graphs:** A great deal of work on keyword querying of structured and semi-structured data has been proposed in recent years. Well known algorithms includes the backward expanding search [3], bidirectional search [21], dynamic programming techniques DPBF [8], and BLINKS [14]. Recently, work that extend keyword search to graphs on external memory has been proposed [7].

This rest of the chapter is organized as follows. We first discuss keyword search methods for schema graphs. In Section 2 we focus on keyword search for XML data, and in Section 3, we focus on keyword search for relational data. In Section 4, we introduce several algorithms for keyword search on schema-free graphs. Section 5 contains a discussion of future directions and the conclusion.

2. Keyword Search on XML Data

Sophisticated query languages such as XQuery have been developed for querying XML documents. Although XQuery can express many queries precisely and effectively, it is by no means a user-friendly interface for accessing XML data: users must master a complex query language, and in order to use it, they must have a full understanding of the schema of the underlying XML data. Keyword search, on the other hand, offers a simple and user-friendly interface. Furthermore, the tree structure of XML data gives nice semantics to the query and enables efficient query processing.

2.1 Query Semantics

In the most basic form, as in XRank [13] and many other systems, a keyword search query consists of n keywords: $Q = \{k_1, \cdots, k_n\}$. XSEarch [6] extends the syntax to allow users to specify which keywords *must* appear in a satisfying document, and which *may* or *may not* appear (although the appearance of such keywords is desirable, as indicated by the ranking function).

Syntax aside, one important question is, what qualifies as an answer to a keyword search query? In information retrieval, we simply return documents that contain all the keywords. For keyword search on an XML document, we want to return *meaningful* snippets of the document that contains the keywords. One interpretation of *meaningful* is to find the *smallest* subtrees that contain all the keywords.

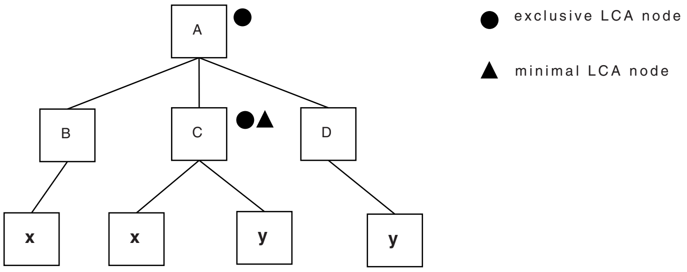

Figure 8.1. Query Semantics for Keyword Search $Q = \{x, y\}$ on XML Data

Specifically, for each keyword k_i, let L_i be the list of nodes in the XML document that contain keyword k_i. Clearly, subtrees formed by at least one node from each $L_i, i = 1, \cdots, n$ contain all the keywords. Thus, an answer to the query can be represented by $lca(n_1, \cdots, n_n)$, the lowest common ancestor (LCA) of nodes $n_1, \cdots, n_n$ where $n_i \in L_i$. In other words, answering the query is equivalent to finding:

$$LCA(k_1, \cdots, k_n) = \{lca(n_1, \cdots, n_n) | n_1 \in L_1, \cdots, n_n \in L_n\}$$

Moreover, we are only interested in the "smallest" answer, that is,

$$\begin{aligned} SLCA(k_1, \cdots, k_n) = \{v \mid v \in LCA(k_1, \cdots, k_n) \ \wedge \\ \forall v' \in LCA(k_1, \cdots, k_n), v \not\prec v'\} \end{aligned} \tag{8.1}$$

where $\prec$ denotes the ancestor relationship between two nodes in an XML document. As an example, in Figure 8.1, we assume the keyword query is $Q = \{x, y\}$. We have $C \in SLCA(x, y)$ while $A \in LCA(x, y)$ but $A \notin SLCA(x, y)$.

Several algorithms including [28, 17, 29] are based on the SLCA semantics. However, SLCA is by no means the only meaningful semantics for keyword search on XML documents. Consider Figure 8.1 again. If we remove node C and the two keyword nodes under C, the remaining tree is still an answer to the query. Clearly, this answer is independent of the answer $C \in SLCA(x, y)$, yet it is not represented by the SLCA semantics.

XRank [13], for example, adopts different query semantics for keyword search. The set of answers to a query $Q = \{k_1, \cdots, k_n\}$ is defined as:

$$\begin{aligned} ELCA(k_1, \cdots, k_n) = \{v \mid \forall k_i\ \exists c\ \ c \text{ is a child node of } v\ \wedge \\ \nexists c' \in LCA(k_1, \cdots, k_n) \text{ and } c \prec c' \wedge \\ c \text{ contains } k_i \text{ directly or indirectly}\} \end{aligned} \tag{8.2}$$

$ELCA(k_1, \cdots, k_n)$ contains the set of nodes that contain at least one occurrence of all of the query keywords, after excluding the sub-nodes that already contain all of the query keywords. Clearly, in Figure 8.1, we have $A \in ELCA(k_1, \cdots, k_n)$. More generally, we have

$$SLCA(k_1, \cdots, k_n) \subseteq ELCA(k_1, \cdots, k_n) \subseteq LCA(k_1, \cdots, k_n)$$

Query semantics has a direct impact on the complexity of query processing. For example, answering a keyword query according to the ELCA query semantics is more computationally challenging than according to the SLCA query semantics. In the latter, the moment we know a node l has a child c that contains all the keywords, we can immediately determine that node l is not an SLCA node. However, we cannot determine that l is not an ELCA node because l may contain keyword instances that are not under c and are not under any node that contains all keywords [28, 29].

2.2 Answer Ranking

It is clear that according to the lowest common ancestor (LCA) query semantics, potentially many answers will be returned for a keyword query. It is also easy to see that, due to the difference of the nested XML structure where the keywords are embedded, not all answers are equal. Thus, it is important to devise a mechanism to rank the answers based on their relevance to the query. In other words, for every given answer tree T containing all the keywords, we want to assign a numerical score to T. Many approaches for keyword search on XML data, including XRank [13] and XSEarch [6], present a ranking method.

To decide which answer is more desirable for a keyword query, we note several properties that we would like a ranking mechanism to take into consideration:

1 *Result specificity*. More specific answers should be ranked higher than less specific answers. The SLCA and ELCA semantics already exclude certain answers based on result specificity. Still, this criterion can be further used to rank satisfying answers in both semantics.

2 *Semantic-based keyword proximity*. Keywords in an answer should appear close to each other. Furthermore, such closeness must reflect the semantic distance as prescribed by the XML embedded structure. Example 8.1 demonstrates this need.

3 *Hyperlink Awareness*. LCA-based semantics largely ignore the hyperlinks in XML documents. The ranking mechanism should take hyperlinks into consideration when computing nodes' authority or prestige as well as keyword proximity.

The ranking mechanism used by XRank [13] is based on an adaptation of $PageRank$ [4]. For each element v in the XML document, XRank defines $ElemRank(v)$ as v's objective importance, and $ElemRank(v)$ is computed using the underlying embedded structure in a way similar to $PageRank$. The difference is that $ElemRank$ is defined at node granularity, while $PageRank$ at document granularity. Furthermore, $ElemRank$ looks into the nested structure of XML, which offers richer semantics than the hyperlinks among documents do.

Given a path in an XML document $v_0, v_1, \cdots, v_t, v_{t+1}$, where v_{t+1} directly contains a keyword k, and v_{i+1} is a child node of v_i, for $i = 0, \cdots, t$, XRank defines the rank of v_i as:

$$r(v_i, k) = ElemRank(v_t) \times decay^{t-i}$$

where $decay$ is a value in the range of 0 to 1. Intuitively, the rank of v_i with respect to a keyword k is $ElemRank(v_t)$ scaled appropriately to account for the specificity of the result, where v_t is the parent element of the value node v_{t+1} that directly contains the keyword k. By scaling down $ElemRank(v_t)$, XRank ensures that less specific results get lower ranks. Furthermore, from node v_i, there may exist multiple paths leading to multiple occurrences of keyword k. Thus, the rank of v_i with respect to k should be a combination of the ranks for all occurrences. XRank uses $\hat{r}(v, k)$ to denote the rank of node v with respect to keyword k:

$$\hat{r}(v, k) = f(r_1, r_2, \cdots, r_m)$$

where $r_1, \cdots, r_m$ are the ranks computed for each occurrence of k (using the above formula), and f is a combination function (e.g., sum or max). Finally, the overall ranking of a node v with respect to a query Q which contains n keywords $k_1, \cdots, k_n$ is defined as:

$$R(v, Q) = \left(\sum_{1 \le i \le n} \hat{r}(v, k_i) \right) \times p(v, k_1, k_2, \cdots, k_n) \qquad (8.3)$$

Here, the overall ranking $R(v, Q)$ is the sum of the ranks with respect to keywords in Q, multiplied by a measure of keyword proximity $p(v, k_1, k_2, \cdots, k_n)$, which ranges from 0 (keywords are very far apart) to 1 (keywords occur right next to each other). A simple proximity function is the one that is inversely proportional to the size of the smallest text window that contains occurrences of all keywords $k_1, k_2, \cdots, k_n$. Clearly, such a proximity function may not be optimal as it ignores the structure where the keywords are embedded, or in other words, it is not a semantic-based proximity measure.

Eq 8.3 depends on function $ElemRank()$, which measures the importance of XML elements bases on the underlying hyperlinked structure. $ElemRank$ is a global measure and is not related to specific queries. XRank [13] defines $ElemRank()$ by adapting PageRank:

$$PageRank(v) = \frac{1-d}{N} + d \times \sum_{(u,v)\in E} \frac{PageRank(u)}{N_u} \tag{8.4}$$

where N is the total number of documents, and N_u is the number of out-going hyperlinks from document u. Clearly, $PageRank(v)$ is a combination of two probabilities: i) $\frac{1}{N}$, which is the probability of reaching v by a random walk on the entire web, and ii) $\frac{PageRank(u)}{N_u}$, which is the probability of reaching v by following a link on web page u.

Clearly, a link from page u to page v propagates "importance" from u to v. To adapt PageRank for our purpose, we must first decide what constitutes a "link" among elements in XML documents. Unlike HTML documents on the Web, there are three types of links within an XML document: importance can propagate through a hyperlink from one element to the element it points to; it can propagate from an element to its sub-element (containment relationship); and it can also propagate from a sub-element to its parent element. XRank [13] models each of the three relationships in defining $ElemRank()$:

$$\begin{aligned} ElemRank(v) = & \frac{1-d_1-d_2-d_3}{N_e} + \\ & d_1 \times \sum_{(u,v)\in HE} \frac{ElemRank(u)}{N_h(u)} + \\ & d_2 \times \sum_{(u,v)\in CE} \frac{ElemRank(u)}{N_c(u)} + \\ & d_3 \times \sum_{(u,v)\in CE^{-1}} ElemRank(u) \end{aligned} \tag{8.5}$$

where N_e is the total number of XML elements, $N_c(u)$ is the number of sub-elements of u, and $E = HE \cup CE \cup CE^{-1}$ are edges in the XML document, where HE is the set of hyperlink edges, CE the set of containment edges, and CE^{-1} the set of reverse containment edges.

As we have mentioned, the notion of keyword proximity in XRank is quite primitive. The proximity measure $p(v, k_1, \cdots, k_n)$ in Eq 8.3 is defined to be inversely proportional to the size of the smallest text window that contains all the keywords. However, this does not guarantee that such an answer is always the most meaningful.

Example 8.1. *Semantic-based keyword proximity*

```
<proceedings>
   <inproceedings>
        <author>Moshe Y. Vardi</author>
        <title>Querying Logical Databases</title>
   </inproceedings>
   <inproceedings>
        <author>Victor Vianu</author>
        <title>A Web Odyssey: From Codd to XML</title>
   </inproceedings>
</proceedings>
```

For instance, given a keyword query "Logical Databases Vianu", the above XML snippet [6] will be regarded as a good answer by XRank, since all keywords occur in a small text window. But it is easy to see that the keywords do not appear in the same context: "Logical Databases" appears in one paper's title and "Vianu" is part of the name of another paper's author. This can hardly be an ideal response to the query. To address this problem, XSEarch [6] proposes a semantic-based keyword proximity measure that takes into account the nested structure of XML documents.

XSEarch defines an *interconnected* relationship. Let n and n' be two nodes in a tree structure T. Let $|n, n'$ denote the tree consisting of the paths from the lowerest common ancestor of n and n' to n and n'. The nodes n and n' are *interconnected* if one of the following conditions holds:

- $T_{|n,n'}$ does not contain two distinct nodes with the same label, or
- the only two distinct nodes in $T_{|n,n'}$ with the same label are n and n'.

As we can see, the element that matches keywords "Logical Databases" and the element that matches keyword "Vianu" in the previous example are not interconnected, because the answer tree contains two distinct nodes with the same label "inproceedings". XSEarch requires that all pairs of matched elements in the answer set are interconnected, and XSEarch proposes an all-pairs index to efficiently check the connectivity between the nodes.

In addition to using a more sophisticated keyword proximity measure, XSEarch [6] also adopts a *tfidf* based ranking mechanism. Unlike standard information retrieval techniques that compute *tfidf* at document level, XSEarch computes the weight of keywords at a lower granularity, i.e., at the level of the leaf nodes of a document. The term frequency of keyword k in a leaf node n_l is defined as:

$$tf(k, n_l) = \frac{occ(k, n_l)}{max\{occ(k', n_l)|k' \in words(n_l)\}}$$

where $occ(k, n_l)$ denotes the number of occurrences of k in n_l. Similar to the standard tf formula, it gives a larger weight to frequent keywords in sparse nodes. XSEarch also defines the inverse leaf frequency (ilf):

$$ilf(k) = \log\left(1 + \frac{|N|}{|\{n' \in N|k \in words(n')|\}}\right)$$

where N is the set of all leaf nodes in the corpus. Intuitively, $ilf(k)$ is the logarithm of the inverse leaf frequency of k, i.e., the number of leaves in the corpus over the number of leaves that contain k. The weight of each keyword $w(k, n_l)$ is a normalized version of the value $tfilf(k, n_l)$, which is defined as $tf(k, n_l) \times ilf(k)$.

With the $tfilf$ measure, XSEarch uses the standard vector space model to determine how well an answer satisfies a query. The measure of similarity between a query Q and an answer N is the sum of the cosine distances between the vectors associated with the nodes in N and the vectors associated with the terms that they match in Q [6].

2.3 Algorithms for LCA-based Keyword Search

Search engines endeavor to speed up the query: find the documents where word X occurs. A word level inverted list is used for this purpose. For each word X, the inverted list stores the id of the documents that contain the word X. Keyword search over XML documents operates at a finer granularity, but still we can use an inverted list based approach: For each keyword, we store all the elements that either directly contain the keyword, or contain the keyword through their descendents. Then, given a query $Q = \{k_1, \cdots, k_n\}$, we find common elements in all of the n inverted lists corresponding to k_1 through k_n. These common elements are potential root nodes of the answer trees.

This na"ve approach, however, may incur significant cost of time and space as it ignores the ancestor-descendant relationships among elements in the XML document. Clearly, for each smallest LCA that satisfies the query, the algorithm will produce all of its ancestors, which may likely be pruned according to the query semantics. Furthermore, the na"ve approach also incurs significant storage overhead, as each inverted list not only contains the XML element that directly contains the keyword, but also all of its ancestors [13].

Several algorithms have been proposed to improve the na"ve approach. Most systems for keyword search over XML documents [13, 25, 28, 19, 17, 29] are based on the notion of lowest common ancestors (LCAs) or its variations. XRank [13], for example, uses the ELCA semantics. XRank proposes two core algorithms, DIL (Dewey Inverted List) and RDIL (Ranked Dewey Inverted List). As RDIL is basically DIL integrated with ranking, due to space considerations, we focus on DIL in this section.

The DIL algorithm encodes ancestor-descendant relationships into the element IDs stored in the inverted list. Consider the tree representation of an XML document, where the root of the XML tree is assigned number 0, and sibling nodes are assigned sequential numbers $0, 1, 2, \cdots, i$. The Dewey ID of a node n is the concatenation of the numbers assigned to the nodes on the path from the root to n. Unlike the na"ve algorithm, in XRank, the inverted list for a keyword k contains only the Dewey IDs of nodes that *directly* contain k. This reduces much of the space overhead of the na"ve approach. From their Dewey IDs, we can easily figure out the ancestor-descendant relationships between two nodes: node A is an ancestor of node B iff the Dewey ID of node A is a prefix of that of node B.

Given a query $Q = \{k_1, \cdots, k_n\}$, the DIL algorithm makes a single pass over the n inverted lists corresponding to k_1 through k_n. The goal is to sort-merge the n inverted lists to find the ELCA answers of the query. However, since only nodes that directly contain the keywords are stored in the inverted lists, the standard sort-merge algorithm cannot be used. Nevertheless, the ancestor-descendant relationships have been encoded in the Dewey ID, which enables the DIL algorithm to derive the common ancestors from the Dewey IDs of nodes in the lists. More specifically, as each prefix of a node's Dewey ID is the Dewey ID of the node's ancestor, computing the longest common prefix will compute the ID of the lowest ancestor that contains the query keywords. In XRank, the inverted lists are sorted on the Dewey ID, which means all the common ancestors are clustered together. Hence, this computation can be done in a single pass over the n inverted lists. The complexity of the DIL algorithm is thus $O(nd|S|)$ where $|S|$ is the size of the largest inverted list for keyword $k_1, \cdots, k_n$ and d is the depth of the tree.

More recent approaches seek to further improve the performance of XRank [13]. Both the DIL and the RDIL algorithms in XRank need to perform a full scan of the inverted lists for every keyword in the query. However, certain keywords may be very frequent in the underlying XML documents. These keywords correspond to long inverted lists that become the bottleneck in query processing. XKSearch [28], which adopts the SLCA semantics for keyword search, is proposed to address the problem. XKSearch makes an observation that, in contrast to the general LCA semantics, the number of SLCAs is bounded by the length of the inverted list that corresponds to the least frequent keyword. The key intuition of XKSearch is that, given two keywords w_1 and w_2 and a node v that contains keyword w_1, there is no need to inspect the whole inverted list of keyword w_2 in order to find all possible answers. Instead, we only have to find the *left match* and the *right match* of the list of w_2, where the left (right) match is the node with the greatest (least) id that is smaller (greater) than or equal to the id of v. Thus, instead of scanning the inverted lists, XKSearch performs an indexed search on the lists. This enables XKSearch to reduce the number of disk accesses to $O(n|S_{min}|)$, where n is the number of the keywords in the query, and S_{min} is the length of the inverted list that corresponds to the least frequent keyword in the query (XKSearch assumes a B-tree disk-based structure where non-leaf nodes of the B-Tree are cached in memory). Clearly, this approach is meaningful only if at least one of the query keywords has very low frequency.

3. Keyword Search on Relational Data

A tremendous amount of data resides in relational databases but is reachable via SQL only. To provide the data to users and applications that do not have the knowledge of the schema, much recent work has explored the possibility of using keyword search to access relational databases [1, 18, 3, 16, 21, 2]. In this section, we discuss the challenges and methods of implementing this new query interface.

3.1 Query Semantics

Enabling keyword search in relational databases without requiring the knowledge of the schema is a challenging task. Keyword search in traditional information retrieval (IR) is on the document level. Specifically, given a query $Q = \{k_1, \cdots, k_n\}$, we employ techniques such as the inverted lists to find documents that contain the keywords. Then, our question is, what is relational database's counterpart of IR's notion of "documents"?

It turns out that there is no straightforward mapping. In a relational schema designed according to the normalization principle, a logical unit of information is often disassembled into a set of entities and relationships. Thus, a relational database's notion of "document" can only be obtained by joining multiple tables.

Naturally, the next question is, can we enumerate all possible joins in a database? In Figure 8.2, as an example (borrowed from [1]), we show all potential joins among database tables $\{T_1, T_2, \cdots, T_5\}$. Here, a node represents a table. If a foreign key in table T_i references table T_j, an edge is created between T_i and T_j. Thus, any connected subgraph represents a potential join.

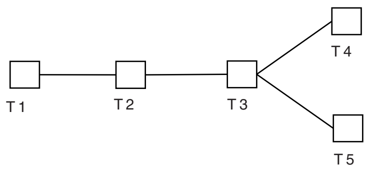

Figure 8.2. Schema Graph

Given a query $Q = \{k_1, \cdots, k_n\}$, a possible query semantics is to check all potential joins (subgraphs) and see if there exists a row in the join results that contains all the keywords in Q.

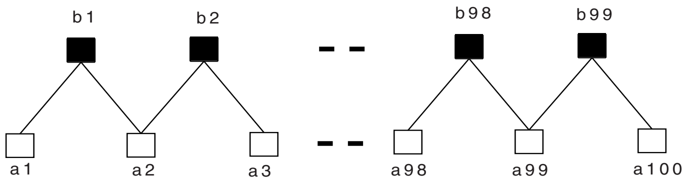

Figure 8.3. The size of the join tree is only bounded by the data Size

However, Figure 8.2 does not show the possibility of self-joins, i.e., a table may contain a foreign key that references the table itself. More generally, the schema graph may contain a cycle, which involves one or more tables. In this case, the size of the join is only bounded by the data size [18]. We demonstrates this issue with a self-join in Figure 8.3, where the self-join is on a table containing tuples (a_i, b_j), and the tuple (a_1, b_1) can be connected with tuple (a_{100}, b_{99}) by repeated self-joins. Thus, the join tree in Figure 8.3 satisfies keyword query $Q = \{a_1, a_{100}\}$. Clearly, the size of the join is only bounded by the number of tuples in the table. Such query semantics is hard to implement in practice. To mitigate this vulnerability, we change the semantics by introducing a parameter K to limit the size of the join we search for answers. In the above example, the result of (a_1, a_{100}) is only returned if K is as large as 100.

3.2 DBXplorer and DISCOVER

DBXplorer [1] and DISCOVER [18] are the most well known systems that support keyword search in relational databases. While implementing the query semantics discussed before, these approaches also focus on how to leverage the physical database design (e.g., the availability of indexes on various database columns) for building compact data structures critical for efficient keyword search over relational databases.

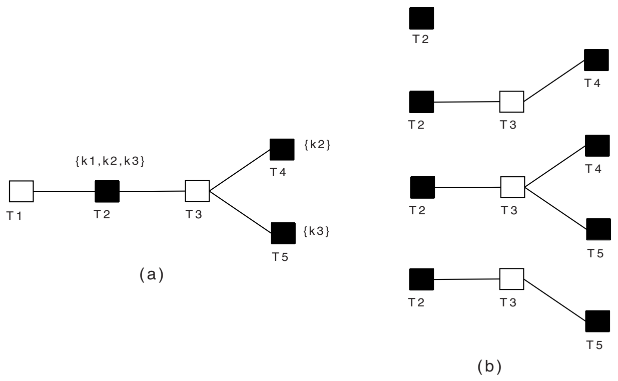

Figure 8.4. Keyword matching and join trees enumeration

Traditional information retrieval techniques use inverted lists to efficiently identify documents that contain the keywords in the query. In the same spirit, DBXplorer maintains a symbol table, which identifies columns in database tables that contain the keywords. Assuming index is available on the column, then given the keyword, we can efficiently find the rows that contain the keyword. If index is not available on a column, then the symbol table needs to map keywords to rows in the database tables directly.

Figure 8.4 shows an example. Assume the query contains three keywords $Q = \{k_1, k_2, k_3\}$. From the symbol table, we find tables/columns that contain one or more keywords in the query, and these tables are represented by black nodes in the Figure: k_1, k_2, k_3 all occur in T_2 (in different columns), k_2 occurs in T_4, and k_3 occurs in T_5. Then, DBXplorer enumerates the four possible join trees, which are shown in Figure 8.4(b). Each join tree is then mapped to a single SQL statement that joins the tables as specified in the tree, and selects those rows that contain all the keywords. Note that DBXplorer does not consider solutions that include two tuples from the same relation, or the query semantics required for problems shown in Figure 8.3.

DISCOVER [18] is similar to DBXplorer in the sense that it also finds all join trees (called candidate networks in DISCOVER) by constructing join expressions. For each candidate join tree, an SQL statement is generated. The trees may have many common components, that is, the generated SQL statements have many common join structures. An optimal execution plan seeks to maximize the reuse of common subexpressions. DISCOVER shows that the task of finding the optimal execution plan is NP-complete. DISCOVER introduces a greedy algorithm that provides near-optimal plan execution time cost. Given a set of join trees, in each step, it chooses the join m between two base tables or intermediate results that maximizes the quantity $\frac{frequency^a}{\log^b(size)}$, where $frequency$ is the number of occurences of m in the join trees, $size$ is the estimated number of tuples of m and a, b are constants. The $frequency^a$ term of the quantity maximizes the reusability of the intermediate results, while the $log^b(size)$ minimizes the size of the intermediate results that are computed first.

DBXplorer and DISCOVER use very simple ranking strategy: the answers are ranked in ascending order of the number of joins involved in the tuple trees; the reasoning being that joins involving many tables are harder to comprehend. Thus, all tuple trees consisting of a single tuple are ranked ahead of all tuples trees with joins. Furthermore, when two tuple trees have the same number of joins, their ranks are determined arbitrarily. BANKS [3] (see Section 4) combines two types of information in a tuple tree to compute a score for ranking: a weight (similar to PageRank for web pages) of each tuple, and a weight of each edge in the tuple tree that measures how related the two tuples are. Hristidis et al. [16] propose a strategy that applies IR-style ranking methods into the computation of ranking scores in a straightforward manner.

4. Keyword Search on Schema-Free Graphs

Graphs formed by relational and XML data are confined by their schemas, which not only limit the search space of keyword query, but also help shape the query semantics. For instance, many keyword search algorithms for XML data are based on the lowest common ancestor (LCA) semantics, which is only meaningful for tree structures. Challenges for keyword search on graph data are two-fold: what is the appropriate query semantics, and how to design efficient algorithms to find the solutions.

4.1 Query Semantics and Answer Ranking

Let the query consist of n keywords $Q = \{k_1, k_2, \cdots, k_n\}$. For each keyword k_i in the query, let S_i be the set of nodes that match the keyword k_i. The goal is to define what is a qualified answer to Q, and the score of the answer.

As we know, the semantics of keyword search over XML data is largely defined by the tree structure, as most approaches are based on the lowest common ancestor (LCA) semantics. Many algorithms for keyword search over graphs try to use similar semantics. But in order to do that, the answer must first form trees embedded in the graph. In many graph search algorithms, including BANKS [3], the bidirectional algorithm [21], and BLINKS [14], a response or an answer to a keyword query is a minimal rooted tree T embedded in the graph that contains at least one node from each S_i.

We need a measure for the "goodness" of each answer. An answer tree T is good if it is meaningful to the query, and the meaning of T lies in the tree structure, or more specifically, how the keyword nodes are connected through paths in T. In [3, 21], their goodness measure tries to decompose T into edges and nodes, score the edges and nodes separately, and combine the scores. Specifically, each edge has a pre-defined weight, and default to 1. Given an answer tree T, for each keyword k_i, we use $s(T, k_i)$ to represent the sum of the edge weights on the path from the root of T to the leaf containing keyword k_i. Thus, the aggregated edge score is $E = \sum_i^n s(T, k_i)$. The nodes, on the other hand, are scored by their global importance or prestige, which is usually based on PageRank [4] random walk. Let N denote the aggregated score of nodes that contain keywords. The combined score of an answer tree is given by $s(T) = EN^\lambda$ where λ helps adjust the importance of edge and node scores [3, 21].

Query semantics and ranking strategies used in BLINKS [14] are similar to those of BANKS [14] and the bidirectional search [21]. But instead of using a measure such as $S(T) = EN^\lambda$ to find top-K answers, BLINKS requires that each of the top-K answer has a different root node, or in other words, for all answer trees rooted at the same node, only the one with the highest score is considered for top-K. This semantics guards against the case where a "hub" pointing to many nodes containing query keywords becomes the root for a huge number of answers. These answers overlap and each carries very little additional information from the rest. Given an answer (which is the best, or one of the best, at its root), users can always choose to further examine other answers with this root [14].

Unlike most keyword search on graph data approaches [3, 21, 14], ObjectRank [2] does not return answer trees or subgraphs containing keywords in the query, instead, for ObjectRank, an answer is simply a node that has high authority on the keywords in the query. Hence, a node that does not even contain a particular keyword in the query may still qualify as an answer as long as enough authority on that keyword has flown into that node (Imagine a node that represents a paper which does not contain keyword *OLAP*, but many important papers that contain keyword *OLAP* reference that paper, which makes it an authority on the topic of *OLAP*). To control the flow of authority in the graph, ObjectRank models *labeled* graphs: Each node u has a label $\lambda(u)$ and contains a set of keywords, and each edge e from u to v has a label $\lambda(e)$ that represents a relationship between u and v. For example, a node may be labeled as a *paper*, or a *movie*, and it contains keywords that describe the paper or the movie; a directed edge from a paper node to another paper node may have a label *cites*, etc. A keyword that a node contains directly gives the node certain authority on that keyword, and the authority flows to other nodes through edges connecting them. The amount or the rate of the outflow of authority from keyword nodes to other nodes is determined by the types of the edges which represent different semantic connections.

4.2 Graph Exploration by Backward Search

Many keyword search algorithms try to find trees embedded in the graph so that similar query semantics for keyword search over XML data can be used. Thus, the problem is how to construct an embedded tree from keyword nodes in the graph. In the absence of any index that can provide graph connectivity information beyond a single hop, BANKS [3] answers a keyword query by exploring the graph starting from the nodes containing at least one query keyword – such nodes can be identified easily through an inverted-list index. This approach naturally leads to a *backward search* algorithm, which works as follows.

1 At any point during the backward search, let E_i denote the set of nodes that we know can reach query keyword k_i; we call E_i the *cluster* for k_i.

2 Initially, E_i starts out as the set of nodes O_i that directly contain k_i; we call this initial set the *cluster origin* and its member nodes *keyword nodes*.

3 In each search step, we choose an incoming edge to one of previously visited nodes (say v), and then follow that edge *backward* to visit its source node (say u); any E_i containing v now expands to include u as well. Once a node is visited, all its incoming edges become known to the search and available for choice by a future step.

4 We have discovered an answer root x if, for each cluster E_i, either $x \in E_i$ or x has an edge to some node in E_i.

BANKS uses the following two strategies for choosing what nodes to visit next. For convenience, we define the distance from a node n to a set of nodes N to be the shortest distance from n to any node in N.

1 ***Equi-distance expansion in each cluster***: This strategy decides which node to visit for expanding a keyword. Intuitively, the algorithm expands a cluster by visiting nodes in order of increasing distance from the cluster origin. Formally, the node u to visit next for cluster E_i (by following edge $u \rightarrow v$ backward, for some $v \in E_i$) is the node with the shortest distance (among all nodes not in E_i) to O_i.

2 ***Distance-balanced expansion across clusters***: This strategy decides the frontier of which keyword will be expanded. Intuitively, the algorithm attempts to balance the distance between each cluster's origin to its frontier across all clusters. Specifically, let (u, E_i) be the node-cluster pair such that $u \notin E_i$ and the distance from u to O_i is the shortest possible. The cluster to expand next is E_i.

He et al. [14] investigated the optimality of the above two strategies introduced by BANKS [3]. They proved the following result with regard to the first strategy, *equi-distance expansion of each cluster* (the complete proof can be found in [15]):

Theorem 8.2. *An optimal backward search algorithm must follow the strategy of equi-distance expansion in each cluster.*

However, the investigation [14] also showed that the second strategy, *distance-balanced expansion across clusters*, is not optimal and may lead to poor performance on certain graphs. Figure 8.5 shows one such example. Suppose that $\{k_1\}$ and $\{k_2\}$ are the two cluster origins. There are many nodes that can reach k_1 through edges with a small weight (1), but only one edge into k_2 with a large weight (100). With distance-balanced expansion across clusters, we would not expand the k_2 cluster along this edge until we have visited all nodes within distance 100 to k_1. It would have been unnecessary to visit many of these nodes had the algorithm chosen to expand the k_2 cluster earlier.

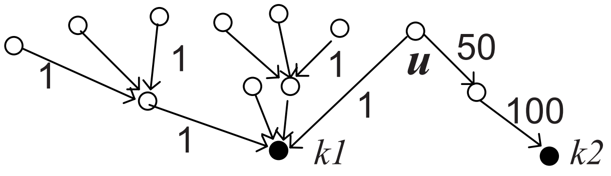

Figure 8.5. Distance-balanced expansion across clusters may perform poorly.

4.3 Graph Exploration by Bidirectional Search

To address the problem shown in Figure 8.5, Kacholia et al. [21] proposed a *bidirectional search* algorithm, which has the option of exploring the graph by following forward edges as well. The rationale is that, for example, in Figure 8.5, if the algorithm is allowed to explore forward from node u towards k_2, we can identify u as an answer root much faster.

To control the order of expansion, the bidirectional search algorithm prioritizes nodes by heuristic *activation factors* (roughly speaking, PageRank with decay), which intuitively estimate how likely nodes can be roots of answer trees. In the bidirectional search algorithm, nodes matching keywords are added to the iterator with an initial activation factor computed as:

$$a_{u,i} = \frac{nodePrestige(u)}{|S_i|}, \forall u \in S_i \tag{8.6}$$

where S_i is the set of nodes that match keyword i. Thus, nodes of high prestige will have a higher priority for expansion. But if a keyword matches a large number of nodes, the nodes will have a lower priority. The activation factor is spreaded from keyword nodes to other nodes. Each node v spreads a fraction μ of the received activation to its neighbours, and retains the remaining $1 - \mu$ fraction.

As a result, keyword search in Figure 8.5 can be performed more efficiently. The bidirectional search will start from the keyword nodes (dark solid nodes). Since keyword node k_1 has a large fanout, all the nodes pointing to k_1 (including node u) will receive a small amount of activation. On the other hand, the node pointing to k_2 will receive most of the activation of k_2, which then spreads to node u. Thus, node u becomes the most activated node, which happens to be the root of the answer tree.

While this strategy is shown to perform well in multiple scenarios, it is difficult to provide any worst-case performance guarantee. The reason is that activation factors are heuristic measures derived from general graph topology and parts of the graph already visited. They do not accurately reflect the likelihood of reaching keyword nodes through an unexplored region of the graph within a reasonable distance. In other words, without additional connectivity information, forward expansion may be just as aimless as backward expansion [14].

4.4 Index-based Graph Exploration – the BLINKS Algorithm

The effectiveness of forward and backward expansions hinges on the structure of the graph and the distribution of keywords in the graph. However, both forward and backward expansions explore the graph link by link, which means the search algorithms do not have knowledge of either the structure of the graph nor the distribution of keywords in the graph. If we create an index structure to store the keyword reachability information in advance, we can avoid aimless exploration on the graph and improve the performance of keyword search. BLINKS [14] is designed based on this intuition.

BLINKS makes two contributions: First, it proposes a new, *cost-balanced* strategy for controlling expansion across clusters, with a provable bound on its worst-case performance. Second, it uses indexing to support forward jumps in search. Indexing enables it to determine whether a node can reach a keyword and what the shortest distance is, thereby eliminating the uncertainty and inefficiency of step-by-step forward expansion.

Cost-balanced expansion across clusters. Intuitively, BLINKS attempts to balance the number of accessed nodes (i.e., the search cost) for expanding each cluster. Formally, the cluster E_i to expand next is the cluster with the smallest cardinality.

This strategy is intended to be combined with the equi-distance strategy for expansion within clusters: First, BLINKS chooses the smallest cluster to expand, then it chooses the node with the shortest distance to this cluster's origin to expand.

To establish the optimality of an algorithm A employing these two expansion strategies, let us consider an optimal "oracle" backward search algorithm P. As shown in Theorem 8.2, P must also do equi-distance expansion within each cluster. The additional assumption here is that P "magically" knows the right amount of expansion for each cluster such that the total number of nodes visited by P is minimized. Obviously, P is better than the best practical backward search algorithm we can hope for. Although A does not have the advantage of the oracle algorithm, BLINKS gives the following theorem (the complete proof can be found in [15]) which shows that A is m-optimal, where m is the number of query keywords. Since most queries in practice contain very few keywords, the cost of A is usually within a constant factor of the optimal algorithm.

Theorem 8.3. *The number of nodes accessed by A is no more than m times the number of nodes accessed by P, where m is the number of query keywords.*

Index-based Forward Jump. The BLINKS algorithm [14] leverages the new search strategy (*equi-distance* plus *cost-balanced* expansions) as well as indexing to achieve good query performance. The index structure consists of two parts.

- **Keyword-node lists** L_{KN}. BLINKS pre-computes, for each keyword, the shortest distances from every node to the keyword (or, more precisely, to any node containing this keyword) in the data graph. For a keyword w, $L_{KN}(w)$ denotes the list of nodes that can reach keyword w, and these nodes are ordered by their distances to w. In addition to other information used for reconstructing the answer, each entry in the list has two fields $(dist, node)$, where $dist$ is the shortest distance between $node$ and a node containing w.

- **Node-keywordmap** M_{NK}. BLINKS pre-computes, for each node u, the shortest graph distance from u to every keyword, and organize this information in a hash table. Given a node u and a keyword w, $M_{NK}(u, w)$ returns the shortest distance from u to w, or ∞ if u cannot reach any node that contains w. In fact, the information in M_{NK} can be derived from L_{KN}. The purpose of introducing M_{NK} is to reduce the linear time search over L_{KN} for the shortest distance between u and w to $O(1)$ time search over M_{NK}.

The search algorithm can be regarded as index-assisted backward and forward expansion. Given a keyword query $Q = \{k_1, \cdots, k_n\}$, for backward expansion, BLINKS uses a cursor to traverse each keyword-node list $L_{KN}(k_i)$. By construction, the list gives the equi-distance expansion order in each cluster. Across clusters, BLINKS picks a cursor to expand next in a round-robin manner, which implements cost-balanced expansion among clusters. These two together ensure optimal backward search. For forward expansion, BLINKS uses the node-keyword map M_{NK} in a direct fashion. Whenever BLINKS visits a node, it looks up its distance to other keywords. Using this information, it can immediately determine if the root of an answer is found.

The index L_{KN} and M_{NK} are defined over the entire graph. Each of them contains as many as $N \times K$ entries, where N is the number of nodes, and K is the number of distinct keywords in the graph. In many applications, K is on the same scale as the number of nodes, so the space complexity of the index comes to $O(N^2)$, which is clearly infeasible for large graphs. To solve this problem, BLINKS partitions the graph into multiple blocks, and the L_{KN} and M_{NK} index for each block, as well as an additional index structure to assist graph exploration across blocks.

4.5 The ObjectRank Algorithm

Instead of returning sub-graphs that contain all the keywords, ObjectRank [2] applies authority-based ranking to keyword search on labeled graphs, and returns nodes having high authority with respect to all keywords. To certain extent, ObjectRank is similar to BLINKS [14], whose query semantics prescribes that all top-K answer trees have different root nodes. Still, BLINKS returns sub-graphs as answers.

Recall that the bidirectional search algorithm [21] assigns activation factors to nodes in the graph to guide keyword search. Activation factors originate at nodes containing the keywords and propagate to other nodes. For each keyword node u, its activation factor is weighted by $nodePrestige(u)$ (Eq. 8.6), which reflects the importance or authority of node u. Kacholia et al. [21] did not elaborate on how to derive $nodePrestige(u)$. Furthermore, since graph edges in [21] are all the same, to spread the activation factor from a node u, it simply divides u's activation factor by u's fanout.

Similar to the activation factor, in ObjectRank [2], authority originates at nodes containing the keywords and flows to other nodes. Furthermore, nodes and edges in the graphs are labeled, giving graph connections semantics that controls the amount or the rate of the authority flow between two nodes.

Specifically, ObjectRank assumes a labeled graph G is associated with some predetermined schema information. The schema information decides the rate of authority transfer from a node labeled u_G, through an edge labeled e_G, and to a node labeled v_G. For example, authority transfers at a fixed rate from a *person* to a *paper* through an edge labeled *authoring*, and at another fixed rate from a *paper* to a *person* through an edge labeled *authoring*. The two rates are potentially different, indicating that authority may flow at a different rate backward and forward. The schema information, or the rate of authority transfer, is determined by domain experts, or by a trial and error process.

To compute node authority with regard to every keyword, ObjectRank computes the following:

- **Rates of authority transfer through graph edges.** For every edge $e = (u \rightarrow v)$, ObjectRank creates a forward authority transfer edge $e^f = (u \rightarrow v)$ and a backward authority transfer edge $e^b = (v \rightarrow u)$. Specifically, the authority transfer edges e^f and e^b are annotated with rates $\alpha(e^f)$ and $\alpha(e^b)$:

$$\alpha(e^f) = \begin{cases} \frac{\alpha(e^f_G)}{OutDeg(u, e^f_G)} & \text{if } OutDeg(u, e^f_G) > 0 \\ 0 & \text{if } OutDeg(u, e^f_G) = 0 \end{cases} \tag{8.7}$$

  where $\alpha(e^f_G)$ denotes the fixed authority transfer rate given by the schema, and $OutDeg(u, e^f_G)$ denotes the number of outgoing nodes from u, of type e^f_G. The authority transfer rate $\alpha(e^b)$ is defined similarly.

- **Node authorities.** ObjectRank can be regarded as an extension to PageRank [4]. For each node v, ObjectRank assigns a global authority $ObjectRank^G(v)$ that is independent of the keyword query. The global $ObjectRank^G$ is calculated using the random surfer model, which is similar to PageRank. In addition, for each keyword w and each node v, ObjectRank integrates authority transfer rates in Eq 8.7 with PageRank to calculate a keyword-specific ranking $ObjectRank^w(v)$:

$$ObjectRank^w(v) = d \times \sum_{e=(u \rightarrow v) or (v \rightarrow u)} \alpha(e) \times ObjectRank^w(u) + \\ + \frac{1-d}{|S(w)|} \tag{8.8}$$

  where $S(w)$ is s the set of nodes that contain the keyword w, and d is the damping factor that determines the portion of ObjectRank that a node transfers to its neighbours as opposed to keeping to itself [4]. The final ranking of a node v is the combination combination of $ObjectRank^G(v)$ and $ObjectRank^w(v)$.

5. Conclusions and Future Research

The work surveyed in this chapter include various approaches for keyword search for XML data, relational databases, and schema-free graphs. Because of the underlying graph structure, keyword search over graph data is much more complex than keyword search over documents. The challenges have three aspects, namely, how to define intuitive query semantics for keyword search over graphs, how to design meaningful ranking strategies for answers, and how to devise efficient algorithms that implement the semantics and the ranking strategies.

There are many remaining challenges in the area of keyword search over graphs. One area that is of particular importance is how to provide a semantic search engine for graph data. The graph is the best representation we have for complex information such as human knowledge, social and cultural dynamics, etc. Currently, keyword-oriented search merely provides best-effort heuristics to find relevant "needles" in this humongous "haystack". Some recent work, for example, NAGA [22], has looked into the possibility of creating a semantic search engine. However, NAGA is not keyword-based, which introduces complexity for posing a query. Another important challenge is that the size of the graph is often significantly larger than memory. Many graph keyword search algorithms [3, 21, 14] are memory-based, which means they cannot handle graphs such as the English Wikipedia that has over 30 million edges. Some reacent work, such as [7], organizes graphs into different levels of granularity, and supports keyword search on disk-based graphs.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[3] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[5] Y. Cai, X. Dong, A. Halevy, J. Liu, and J. Madhavan. Personal information management with SEMEX. In *SIGMOD*, 2005.

[6] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.

[7] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. In *VLDB*, pages 1189–1204, 2008.

[8] B. Ding, J. X. Yu, S. Wang, L. Qing, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[9] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.

[10] S. Dumais, E. Cutrell, JJ Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *SIGIR*, 2003.

[11] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Comput. Networks*, 33(1-6):119–135, 2000.

[12] J. Graupmann, R. Schenkel, and G. Weikum. The spheresearch engine for unified ranked retrieval of heterogeneous XML and web documents. In *VLDB*, pages 529–540, 2005.

[13] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.

[14] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searches on graphs. In *SIGMOD*, 2007.

[15] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searches on graphs. Technical report, Duke CS Department, 2007.

[16] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[17] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in XML trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):525–539, 2006.

[18] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[19] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378, 2003.

[20] Haoliang Jiang, Haixun Wang, Philip S. Yu, and Shuigeng Zhou. GString: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[21] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[22] G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[23] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, pages 779–790, 2004.

[24] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[25] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, pages 72–83, 2004.

[26] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[27] Dennis Shasha, Jason T.L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[28] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.

[29] Yu Xu and Yannis Papakonstantinou. Efficient LCA based keyword search in XML data. In *EDBT*, pages 535–546, New York, NY, USA, 2008. ACM.

[30] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.