Efficient Keyword Search Across Heterogeneous Relational Databases

Mayssam Sayyadian[1], Hieu LeKhac[2], AnHai Doan[1], Luis Gravano[3]
[1]University of Wisconsin-Madison [2]University of Illinois-Urbana [3]Columbia University

Abstract

Keyword search is a familiar and potentially effective way to find information of interest that is "locked" inside relational databases. Current work has generally assumed that answers for a keyword query reside within a single database. Many practical settings, however, require that we combine tuples from multiple databases to obtain the desired answers. Such databases are often autonomous and heterogeneous in their schemas and data. This paper describes Kite*, a solution to the keyword-search problem over heterogeneous relational databases.* Kite *combines schema matching and structure discovery techniques to find approximate foreign-key joins across heterogeneous databases. Such joins are critical for producing query results that span multiple databases and relations.* Kite *then exploits the joins – discovered automatically across the databases – to enable fast and effective querying over the distributed data. Our extensive experiments over real-world data sets show that (1) our query processing algorithms are efficient and (2) our approach manages to produce high-quality query results spanning multiple heterogeneous databases, with no need for human reconciliation of the different databases.*

1 Introduction

A vast amount of current data resides in relational databases at enterprises, government agencies, research organizations, and on the PCs of home users. As such, the data is often "locked away," reachable only via SQL query interfaces. To facilitate access to this data, recent work has studied the problem of keyword search over relational databases (e.g., [4, 1, 11, 10, 13, 14, 3]). Such keyword search facilities allow users to query the databases quickly, with no need to know SQL or the database schemas. In addition, keyword search can help discover unexpected answers that are often difficult to obtain via rigid-format SQL queries. The following example illustrates these issues.

Example 1.1 *Consider the simplified database in Figure 1, which belongs to the Service department of a company, with two tables,* Customers *and* Complaints*, listing customer information and their complaints about services, respectively. Suppose a department manager wants to know about the past interaction between an employee named Michael Smith and Cisco. For this, the manager can quickly issue the keyword query [Michael Smith Cisco] to obtain a ranked list of answers. An answer would show that the two tuples t_1 and u_1 contain the query keywords and relate via foreign-key join* cust-id = id*, suggesting that Cisco has made a complaint about a Michael Smith. Another answer would show two tuples t_1 and u_3 (again related via the same join), suggesting that Michael Smith is also involved in another complaint made by Cisco (against John). It would be challenging to write a SQL query to uncover all such potentially interesting connections between Michael Smith and Cisco, because this query would need to check for the occurrence of such keywords in all attributes, and combine these occurrences in all possible meaningful ways.* □

SERVICE DEPARTMENT DATABASE

Customers

tuple-id	cust-id	name	contact	address
t1	c124	Cisco	Michael Jones	1014 W. Main St, Baltimore, MD
t2	c533	IBM	David Long	503 Lincoln Ave, Paris, Texas

Complaints

tuple-id	id	service-id	emp-name	comments
u1	c124	020401	Michael Smith	Line repair didn't work …
u2	c355	130402	Bruce Mayer	Appeared impolite ...
u3	c124	070401	John	Late, deferred work to Michael Smith …
u4	c124	120403	Smith	Overcharged for service …

Figure 1. Sample database with textual relation attributes

Keyword search over relational databases thus provides an attractive querying platform, and has consequently generated substantial research interest. So far, current work on this topic has focused on how to search over a *single* relational database. In practice, however, we often must query *multiple* databases to obtain the desired information.

Example 1.2 *Consider again the service company mentioned earlier. Suppose a manager wants to send an employee named Jack Lucas to Cisco to negotiate a long-term service contract. To ensure a smooth negotiation, the manager wants to know if Jack Lucas has been related in any way to Cisco. To do so, the manager pulls in the database of the Service department (Figure 2.a) and that of the Human Resource department (Figure 2.b), then issues the keyword query [Jack Lucas Cisco] over the collection of the two databases. This query produces an answer (Figure 2.c) that reveals that Jack Lucas manages Michael Smith in a group, and that Cisco has made complaints about a Michael Smith. This information can help the manager decide if Jack Lucas is the right choice, or as preparation for the negotiation. Notice that this information cannot be obtained from each database in isolation.* □

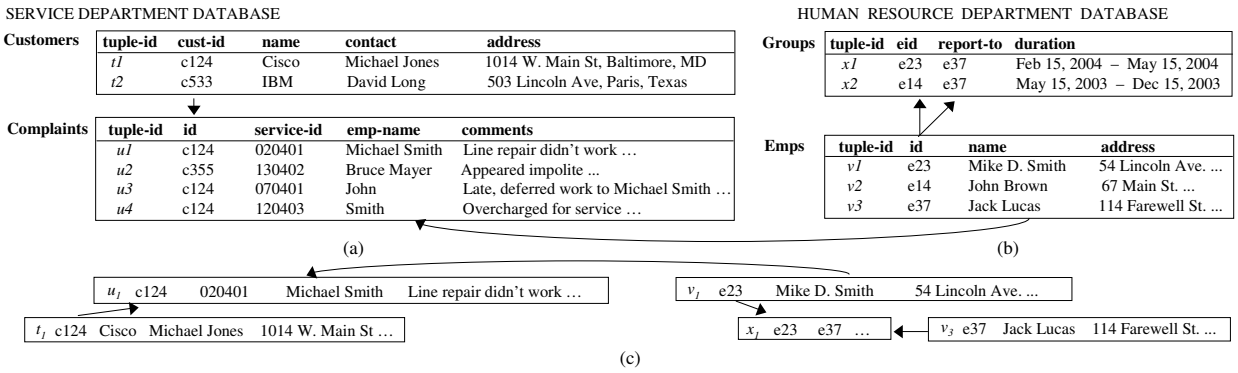

Figure 2. A keyword search across multiple databases

Other examples of the need for keyword search over multiple databases arise naturally. As these examples show, the ability to perform keyword searches over multiple databases is important in many practical settings, and will become increasingly so as the number of such databases grows.

In this paper, we describe Kite, a solution to the keyword-search problem over heterogeneous relational databases. As a key challenge to develop Kite, databases for the potentially dynamic scenarios that we consider have often not been integrated and exhibit *semantic heterogeneity*, at both schema and data levels (e.g., employee names can be referred to as emp-name and name, and Michael Smith can be referred to as "Michael Smith" and "Mike D. Smith" in different databases) [22]. *Manually* integrating such heterogeneous databases is well known to be difficult and might take weeks or months to accomplish [22, 6]. Furthermore, many keyword queries express *ad-hoc, short-term* information needs, and hence they require only a *temporary* assembling of several databases. To address this problem, Kite *automatically* discovers approximate foreign-key joins across heterogeneous databases, since such joins are critical for producing query results that span multiple relations. Kite employs a combination of structure discovery and schema matching methods that empirically outperforms current join discovery algorithms.

After database integration, Kite faces the challenge of searching an often large space of potential query results, to quickly find the top few results for a user query. Searching this space in a multi-database setting is fundamentally much harder than in a single-database setting, for the following reasons. First, the search space grows exponentially with the number of databases and their associated (automatically discovered) foreign-key joins. To address this problem, Kite "condenses" the search space and operates at a higher level of abstraction than do single-database keyword search solutions. Second, answering queries in this multi-database scenario often requires executing foreign-key joins across databases, a much more expensive proposition than over a single database because of communication costs. This increased cost renders single-database exploration strategies ineffective in multi-database settings, thus requiring Kite to develop better exploration strategies that consider the high cost of cross-database joins. Finally, current single-database solutions rely on certain statistics (e.g., the estimated result size of a SQL query [10]) to choose an exploration strategy effectively. Unfortunately, it is often difficult to estimate such statistics accurately in multi-database settings. To address this limitation, Kite develops a novel *adaptive* solution for selecting strategies, which monitors and changes exploration strategies *on-the-fly*, whenever the current strategy no longer appears effective.

In the rest of the paper, we define the problem of keyword search across heterogeneous relational databases and describe our solution, Kite, in detail. We report extensive experimental results over real-world data sets, suggesting that Kite is efficient and produces high-quality query results spanning multiple databases, with no need for manual reconciliation of the different databases.

2 Related Work

Many research efforts have studied the problem of keyword search over a single relational database. Examples include BANKS [4], DBXplorer [1], Discover [11] and more [10, 14, 3, 23, 16] (see also Section 3). Beyond the relational context, keyword search over XML data has attracted attention (e.g., [17, 2, 9]), but these efforts do not consider search scenarios with *multiple* XML databases.

Numerous solutions on data instance matching, as well as many semi-automatic tools for schema matching, have also been proposed (see [22, 6] for surveys). Once such a tool has predicted matches, users typically must *manually* verify and correct these matches before querying can be carried out [22]. In this paper, we focus on practical settings where it is not realistic to assume that the users will have the time or expertise to manually verify the matches. As we will see, we show that *automatic schema matching* is still useful, and that the ranking of query results helps circumvent the inherent imperfection of automatic matching.

Keyword search in peer-to-peer contexts has also received attention recently (e.g., [21, 20, 26, 15]). Such set-

tings commonly involve hundreds or thousands of databases that can leave or join the network at will. Hence these efforts have focused on database selection and distributed indexing [15]. In contrast, we focus on automatically reconciling database heterogeneity and on efficiently finding query results that span multiple databases.

The problem of processing "top-k" queries has attracted recent attention in a number of different scenarios. The design of the top-k searcher that we propose in this paper faces challenges that are related to other top-k query processing work (e.g., [7, 18, 24]). Reference [10] also applies some of the top-k query processing ideas to the problem of keyword search, but for single-database settings.

3 Problem Definition

We now define the problem of keyword search over multiple databases. We consider common settings with a relatively small number of databases (up to the tens), such as the examples discussed in the Introduction. Such settings are pervasive in enterprises and government agencies, and for scientific collaboration and home usage. In contrast, we do not consider (e.g., peer-to-peer) settings with hundreds or thousands of databases. These settings raise additional challenges, including database selection and distributed indexing, and are the subject of interesting future research.

We focus on the realistic scenario where the databases are physically disparate, can be frequently modified, and are often assembled for keyword search in unforeseen ways. Hence, we assume that the database contents cannot be retrieved and "warehoused" in a single central location. However, we do assume that (1) the databases can be queried using standard information retrieval (IR) indexes on the textual attributes [25], and (2) the databases fully cooperate and participate in the execution of our keyword search strategies (e.g., allowing for the creation of the indexes and auxiliary relations, see Section 5).

Single-database search: Before defining the problem of searching over multiple databases, we briefly review the single-database scenario to introduce some necessary concepts. Given a keyword query Q over a relational database D, most keyword-search solutions (e.g., [4, 1, 11, 10]) define an *answer* to Q (also called *tuple trees* in [11, 10]) to be a set of tuples from D connected via foreign-key joins (henceforth *FK joins*, for short). Under *Boolean-AND semantics* [4, 1, 11], the tuples in an answer to Q are required to include *all* keywords in Q. For example, given query Q = [Michael Smith Cisco] over the database in Figure 1, a possible answer is $t_1 \rightarrow u_1$, which contains "Cisco" in t_1 and "Michael Smith" in u_1, and t_1 and u_1 are combined via FK join cust-id = id. Under *Boolean-OR semantics* [10], an answer may cover only a *subset* of the query keywords. Thus answer $t_1 \rightarrow u_1$ is acceptable, and so is $t_1 \rightarrow u_4$, with only two words, "Smith" and "Cisco". The result to query Q is usually a *ranked list* of answers, where the *score* for an answer is inversely proportional to the number of joins in the answer. Early "binary" scoring strategies focused on just the presence or absence of keywords [11]. Subsequently, IR-style TF-IDF scoring was introduced for this problem [10, 16] (see also [4, 3]). Finally, since users often examine only a few answers, recent work [10] has focused on returning the *top-k answers* for Q, for moderate values of k.

The ideal scenario for multi-database search: We now define what it means to search multiple databases with a keyword query Q. We define the *ideal top-k result* for Q in a two-step process. First, we manually integrate the databases, by identifying FK joins across the databases and resolving data instance discrepancies. For example, for the "Service" and "Human Resource" databases in Figures 2.a-b, we may discover that attribute Complaints.emp-name of database "Service" and attribute Emps.name of database "Human Resource" form a FK join, and that "Michael Smith" of Complaints.emp-name matches "Mike D. Smith" of Emps.name. In the second step, we then process query Q over the integrated database to produce the top-k results (e.g., following the IR-style algorithms in [10]). The results of a query may then span multiple databases and involve both "native" FK joins, defined as part of the schema of a database, as well as "derived" FK joins, identified during database integration and involving multiple databases.

Approximating the ideal scenario: Manually integrating databases is labor intensive [22], and thus is prohibitively expensive for our dynamic keyword search settings. Hence, we approximate the ideal scenario by employing automatic solutions to discover FK joins and to match data instances across databases (see Sections 4 and 5).

Once we have automatically identified a set of FK joins across databases, we can generate answers to a keyword query Q just like in the ideal scenario. However, observe that automatic solutions to identify FK joins and to match data instances are inherently imperfect and often produce results only with some confidence score [22]. Hence, we must factor such scores into the answer score. Specifically, let T be an answer to Q, joining tuples from one or more databases. Let $a_1, \ldots, a_n$ be the attributes in T, and $j_1, \ldots, j_m$ be the FK joins used to build T. Furthermore, let $d_1, \ldots, d_m$ be the attribute value pairs "matched" in joins $j_1, \ldots, j_m$, respectively. Then we define the score of T for Q, $score(T, Q)$, as:

$$\frac{\alpha_w \cdot score_w(T, Q) + \alpha_j \cdot score_j(T) + \alpha_d \cdot score_d(T)}{size(T)} \quad (1)$$

where α_w, α_j, and α_d are coefficients, and $size(T)$ is the number of joins in T. Furthermore, (1) $score_w(T, Q) = \sum_{a_i} score(a_i, Q)$, where $score(a_i, Q)$ quantifies how well

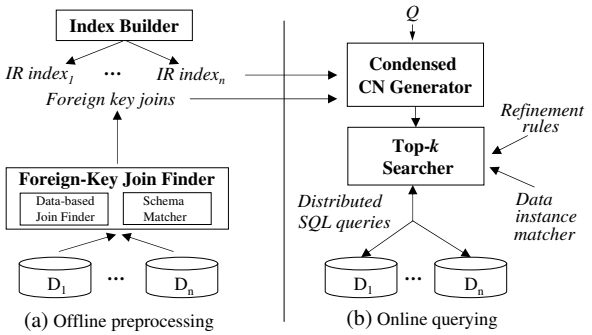

Figure 3. The Kite architecture

attribute a_i in T matches keywords in Q; this score is computed using a TF-IDF formula as shown in Equation 1 of [10]. (2) $score_j(T) = \sum_{j_i} score(j_i)$, where $score(j_i)$ measures the confidence in join j_i of T. If j_i is a FK join *within* a single database, then this confidence is 1; otherwise the confidence is computed as detailed in Section 4. (3) $score_d(T) = \sum_{d_i} score(d_i)$, where $score(d_i)$ measures the "confidence" with which the attribute value pair associated with FK join j_i matched.

In the absence of any further knowledge, we can weight the three terms in (1) equally, as we currently do in Kite. Section 6 shows that this setting works well on the evaluated real-world data sets. More sophisticated schemes could set the coefficients using user-provided relevance feedback.

Problem definition: We can now define the keyword search problem considered in this paper. Given databases $D_1, \ldots, D_n$, a keyword query Q, and a scoring function as defined above, effectively produce the top-k answers for Q from $D_1, \ldots, D_n$, such that these answers closely approximate the ideal top-k result for Q, as defined above.

The rest of the paper describes the Kite solution to this problem. Kite operates in two phases: offline preprocessing and online querying. In the *offline preprocessing phase* (Figure 3.a), the index builder constructs standard inverted IR indexes on the text attributes of the databases. Then, the FK join finder leverages data-based join discovery and schema matching methods to identify FK joins across the databases. In the *online querying phase*, given a top-k keyword query Q, the condensed candidate network (CN) generator employs the FK joins and the IR indexes to quickly identify a space of possible answers to Q. The searcher then explores this space (via SQL queries issued to the databases) to find the top-k answers. In doing so, the searcher employs a set of refinement rules, encoding different exploration strategies, and a data instance matcher.

The next section describes the FK join finder. Section 5 then describes the index builder, the condensed CN generator, and the top-k searcher.

4 Joins Across Multiple Databases

As discussed earlier, a key challenge to process keyword queries over multiple databases is to discover FK joins. Kite employs data-based key and join discovery algorithms [12, 5] to find FK joins. Then, Kite prunes the set of discovered FK joins using a schema matching method [19]. We found that adding the pruning step with schema matching can greatly improve the accuracy of FK join discovery (by 15-49% in our experiments), which is significant because incorrect FK joins can substantially increase the size of the search space for the top-k searcher, as well as decrease the quality of the answers produced.

To explain Kite's join discovery module, consider two tables U and V that belong to different databases. Our goal is to find all FK joins in V that reference a key of table U. For this, we first find all keys in U, since they will participate in any FK joins that we discover. Then, we consider each key of U individually, and identify any attribute sets in V that could be meaningfully joined with the key. Next, we generate candidate FK joins. Finally, we only keep candidates that are "semantically correct," as we discuss below:

1. Finding keys in table U: We cannot just rely on the schema-defined keys of table U, because some of these keys may not be helpful for participating in FK joins across databases. For example, an id attribute of U might be meaningless to join with a table V in some other database, because databases may not share the same id space. Rather than discovering or exploring true keys such as id above, we focus on finding "approximate" keys in U that help in defining appropriate FK joins. For this, we employ an approximate key discovery algorithm developed in [12].

2. Finding joinable attributes in V: Once we have found the approximate keys of U, we find attributes in V that can be joined with these keys. Specifically, for each attribute a in an approximate key of U, we find all attributes b in V such that a and b are *joinable*, in that they share many similar values. To execute this step efficiently, we employ Bellman [5], a state-of-the-art join discovery algorithm that computes statistical synopses for attributes to quickly find "joinable" attributes in large databases.

3. Generating FK join candidates: Next, we identify candidate FKs by exhaustively listing all possible alignments of the key attributes in U with their joinable counterparts in V. As an example, consider a key $\{a_1, a_2\}$ in U and suppose that attribute a_1 is joinable with attribute b_1 of V, while attribute a_2 is joinable with both attributes c_1 and c_2 of V. Then we list two candidate FK joins, J_1: $(b_1, c_1) - (a_1, a_2)$, meaning that attributes (b_1, c_1) of V reference attributes (a_1, a_2) of U, and J_2: $(b_1, c_2) - (a_1, a_2)$.

4. Removing semantically incorrect candidates: Not all candidate FK joins are meaningful, since current join discovery algorithms (e.g., [5, 12]) examine only the similarity

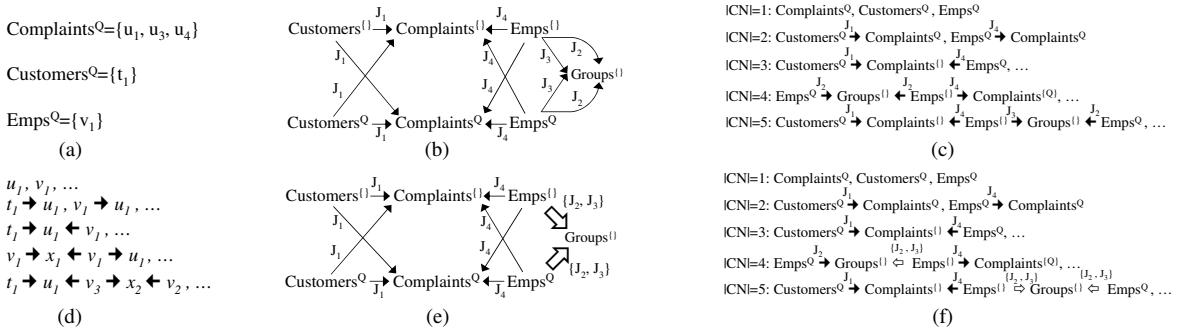

Figure 4. (a) Tuple sets, (b) a tuple set graph, (c) CNs, (d) answers, (e) a condensed tuple set graph, and (f) CCNs in a multi-database

of *data values* to produce join candidates such as J_1 and J_2 in our example above. In fact, attributes may share similar values and yet not be semantically joinable, as is the case for last-name and city-name (both with string values).

To remove spurious candidate foreign keys, we introduce a schema matching step that examines the database *schemas* to find semantically *related* attributes. We then keep only join candidates with semantically related attributes. For example, consider join candidate J_1: $(b_1, c_1) - (a_1, a_2)$. We discard J_1 if either b_1 is found not to match a_1 or c_1 is found not to match a_2 by a schema matching algorithm. Virtually any effective schema matching algorithm [22] can be used in this step. Currently, we employ the publicly available Simflood algorithm [19], which matches attributes based on the similarity of their names and neighboring attributes. We return all the surviving FK joins among all relation pairs across the databases. Note that we focus on discovering "full" FK joins and ignore partial matches where only some but not all of the key attributes of a relation are joinable with attributes of another relation.

5 Scalable Search Across Multiple Databases

We have described how Kite discovers FK joins across the databases $D_1, \ldots, D_n$. Conceptually, $D_1, \ldots, D_n$ together with the discovered joins can be viewed as a single "integrated" database $\mathcal{D}$, whose tables are the tables of $D_1, \ldots, D_n$, and whose FK joins are the native FK joins of these databases as well as the discovered FK joins. We now describe how Kite applies the condensed CN generator and the top-k searcher to $\mathcal{D}$, to produce top-k answers to user queries. We then discuss why current keyword search algorithms over a single database do not scale well over $\mathcal{D}$, thereby highlighting the key innovations of Kite.

5.1 Generating Condensed Candidate Networks

Given a keyword query Q over the integrated database $\mathcal{D}$, Kite starts by creating a set of so-called *candidate networks (CNs)*, each of which specifies a set of answers to Q. CNs have been used extensively for keyword search over a single database [1, 11, 10]. Kite however modifies the definition and generation of CNs, to cope with the exploding search space in multi-database settings. We will first review a current CN generation algorithm (e.g., as employed in [1, 11, 10]), and then we will highlight its limitations, which motivate Kite's solution.

Creating tuple sets: Given a query Q, the CN generation algorithm first searches each table R in $\mathcal{D}$ (using appropriate inverted indexes) to find all tuples that contain some keywords in Q. These tuples form a *tuple set*, denoted as R^Q. For example, let $\mathcal{D}$ consist of the "Service" and "Human Resource" databases in Figures 2.a-b, and Q = [Smith Cisco]. Then, the algorithm generates the three tuples sets shown in Figure 4.a. The first set, $\mathsf{Complaints}^Q$, consists of tuples u_1, u_3, and u_4 of table Complaints, because these tuples contain keyword "Smith" (see Figure 2.a).

Creating a tuple set graph: Next, the CN generation algorithm uses the tuple sets, the schemas of the individual databases, and the discovered FK joins to construct a *tuple set graph* (Figure 4.b), which compactly specifies all the possible ways that tuples in tuple sets can be linked to each other via FK join paths, either *within* or *across* databases. For example, the path $\mathsf{Customers}^Q \rightarrow \mathsf{Complaints}^{\{\}} \leftarrow \mathsf{Emps}^Q$ in this graph (see Figure 4.b) specifies that a tuple in $\mathsf{Customers}^Q$ may be linked to a tuple in Emps^Q via some tuple in Complaints. The notation $\mathsf{Complaints}^{\{\}}$ signifies that Complaints serves as a "bridging" relation in this case.

Creating CNs: Finally, the CN generation algorithm searches the tuple set graph to create *trees* with certain properties, such as not exceeding a prespecified size (see [1, 11, 10]). Figure 4.c shows examples of trees of various sizes. Each tree, together with the associated tuple sets, forms a CN, which specifies a set of answers to Q that can be viewed as conforming to a tree *template*. This set of answers can be obtained by executing a SQL query that "materializes" the CN. For instance, the CN $\mathsf{Customers}^Q \stackrel{J_1}{\rightarrow} \mathsf{Complaints}^Q$ specifies answers such that each links a tuple in $\mathsf{Customers}^Q$ with a tuple in $\mathsf{Complaints}^Q$ via join J_1; the SQL query for these answers is:

```
SELECT    *
FROM      Customers C, Complaints P
WHERE     C.cust-id = P.id AND C.tuple-id = t1 AND
          (P.tuple-id = u1 OR P.tuple-id = u3 OR P.tuple-id = u4)
```

Such SQL queries are frequently executed by the top-k searcher during query processing.

Creating "condensed" CNs in Kite: In multi-database settings, the above CN generation algorithm often generates an unmanageable number of CNs, which makes both CN generation and the subsequent search for top-k answers extremely inefficient. The main reason behind this problem is that, as the number of databases grows, the tuple set graph size grows significantly, and the number of candidate subgraphs that must be considered for CN generation grows exponentially in the number of edges, i.e., FK joins, in the tuple set graph.

Thus, the current CN generation algorithm [1, 11, 10] does not scale well to multi-database settings. To address this limitation, we observe that many CNs often share the same tuple sets and differ only in the associated joins. Kite's solution, then, is to group such CN candidates and treat them as a single "condensed" CN. Specifically, Kite first condenses the tuple set graph by collapsing all joins that combine the same two tuple sets into a single composite join. Figure 4.e shows the condensed version of the tuple set graph in Figure 4.b, where the two edges J_2 and J_3 between Emps$^{\{\}}$ and Groups$^{\{\}}$ have now been condensed into a single edge. Kite then searches for CNs on the simpler condensed tuple set graph. Figure 4.f lists some CNs generated from the condensed tuple set graph of Figure 4.e. We refer to both condensed CNs and "regular" CNs as *Condensed CNs (CCNs)*. By condensing tuple set graphs and generating CCNs, Kite drastically reduces query execution time without compromising result quality, as we will see in Section 6.2.

5.2 Iterative Refinement Search

We have described how Kite generates the CCNs for a query Q, which together encode a typically large space of answers to Q. Kite then performs an *iterative refinement search* in this space to find the top-k answers. Specifically, Kite views each answer to Q as a *concrete state*. A set of concrete states, described in a compact way, forms an *abstract state*. For example, a CCN is an abstract state. Kite associates with each state a *score interval*. The score interval of an abstract state S tightly covers the scores of all concrete states of S, while the score interval of a concrete state is just a single value, namely the state score.

Kite starts with the set of CCNs generated in the previous step (Section 5.1), treating each CCN as an abstract state. Kite then iteratively refines the abstract states into less-abstract or concrete states, computes the state scores, and eliminates suboptimal states, until the algorithm finds the top-k concrete states. Kite thus achieves computational savings by avoiding an exhaustive search of the entire space of answers. An example illustrates the search process:

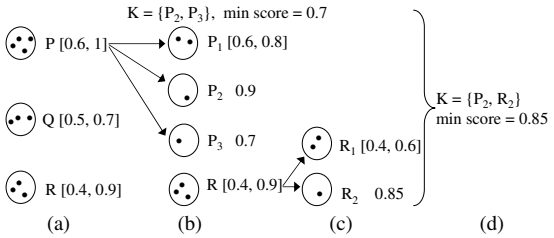

Figure 5. Iterative refinement search in Kite

Example 5.1 *Consider the execution of a top-2 query in Figure 5.a, where P, Q, and R are abstract states. State P consists of four concrete states (denoted with dots) and has a score interval [0.6,1], meaning that the scores of the four concrete states of P lie in this range. To continue processing the query,* Kite *selects P to refine into states P_1, P_2, and P_3 (we will discuss how to select and refine states shortly). Next,* Kite *computes the scores of the new states and eliminates suboptimal states. Figure 5.b shows the remaining states. Note that P_2 and P_3 are concrete states, and hence are also listed in an "accumulator" K that maintains the list of top-2 concrete states found so far. Note also that Q has been eliminated: no concrete state in Q can be among the top-2 states, since K already contains two concrete states, P_2 and P_3, whose minimum score (0.7) is greater than or equal to the upper bound (0.7) on the score interval of Q. Next, suppose that* Kite *selects R and refines it into states R_1 and R_2, shown in Figure 5.c with recomputed scores. R_2 is a concrete state with score 0.85. This score is higher than the score of P_3 (0.7), which is kept in accumulator K. Hence,* Kite *updates accumulator K to contain P_2 and R_2, with a revised minimum score of 0.85. Next,* Kite *eliminates all other states because their score upper bounds are lower than 0.85.* Kite *then returns P_2 and R_2 as the top-2 answers.*□

As described, Kite relies on a small set of crucial decisions: Which state should it choose to refine in each iteration? What is the set of refinement rules that it can use? And which refinement rules should it apply under what conditions? We now elaborate on these decisions.

(a) Selecting a state for refinement: In each iteration, Kite selects for refinement the abstract state S with the *highest* score upper bound. Intuitively, it is not possible to eliminate S without refinement and reach a solution for the query, hence we must refine S. This state selection strategy minimizes the number of states that must be refined, which is desirable because state refinement usually is the most time-consuming operation of the search process and requires executing SQL queries that often span multiple distributed databases.

(b) Defining refinement rules: Kite employs three refinement rules, Full, Partial, and Deep, to refine an abstract state S. Rules Full and Partial are an adaptation of existing single-database strategies [10] to our multi-database

Input: Abstract state S with tuple sets $TS_1 \ldots TS_n$ and composite joins $J_i=\{J_i^1, J_i^2,\ldots\}, \ldots, J_m=\{J_m^1, J_m^2,\ldots\}$
Output: Concrete states $CS_1 \ldots CS_m$, abstract state S
Require: Each tuple set TS_i has a list of *marked_tuples* and *unmarked_tuples* for every join in which it participates
Marked and unmarked tuples in TS_i are sorted in decreasing order of score
Marked and unmarked joins in each composite join are sorted similarly

1. If $\exists\ TS_i$ such that for every J_j: TS_i.*unmarked_tuples*(J_j) = $\varnothing$ then return $\varnothing$
2. Tuple set $TS^*(J^*) = argmax_{i,j}$ (TS_i.*unmarked_tuples*(J_j).*next_tuple*().*score*())
3. Tuple $t = TS^*$.*unmarked_tuples*(J^*).*next_tuple*()
4. Move tuple t from TS^*.*unmarked_tuples*(J^*) to TS^*.*marked_tuples*(J^*)
5. Concrete states $CS_{j=1..m}$=*join*(TS_1.*marked_tuples*(J_j),... ,t,...,TS_n.*marked_tuples*(J_j))
6. Return $CS_1, \ldots, CS_m$, S

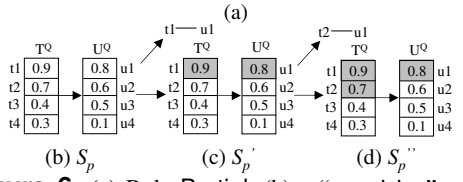

Figure 6. (a) Rule Partial, (b) a "promising" state S_p, and (c)-(d) applying Partial to pull out two concrete states from S_p

scenario with condensed CNs.

Rule Full refines S into *all* constituent concrete states, by executing an appropriate SQL query, as discussed earlier. Full completely materializes all concrete states represented by S. In contrast, Rule Partial, whose pseudocode is in Figure 6.a, refines S only partially, by focusing on a CCN, S_p, with the most "promising" score. Specifically, Partial starts by building S_p from S using the confidence score (see Section 4) of the FK joins in S; for each composite edge in S representing multiple joins, Partial builds S_p from S by just keeping the highest-confidence join. Partial also builds a CCN S_r encoding all remaining states in $S - S_p$, and returns S_p and S_r as the output of the refinement step for S. For example, consider a CCN $S = T^Q \overset{\{J_1,J_2,J_3\}}{\longrightarrow} U^Q$, where tuple sets T^Q and U^Q are linked by a composite edge that represents joins J_1, J_2, and J_3. Furthermore, suppose that the confidence scores for these FK joins are 0.8, 0.6, and 0.5, respectively. Then Partial builds S_p by choosing the highest-confidence join, J_1, so $S_p = T^Q \overset{J_1}{\rightarrow} U^Q$; correspondingly, S_r represents the "residual" states from S not covered by S_p, so $S_r = T^Q \overset{\{J_2,J_3\}}{\longrightarrow} U^Q$.

After exploiting the FK join confidence scores to define S_p, Partial refines S_p further by prioritizing the tuples in the S_p tuple sets by their score, and evaluating only a small "prefix" of these ordered tuple lists; the contributing tuples are *marked* accordingly. The following example illustrates the process (see Figure 6.a for the pseudocode for Partial):

Example 5.2 *To refine the state $S_p = T^Q \overset{J_1}{\rightarrow} U^Q$ mentioned earlier,* Partial *first sorts the tuples in T^Q and U^Q in decreasing order of their score, as shown in Figure 6.b.* Partial *then selects the top two tuples t_1 and u_1 (i.e., those with highest scores) from T^Q and U^Q, respectively, to form a concrete state. If these two tuples join, then* Partial *creates the concrete state $t_1 \rightarrow u_1$. Intuitively,* Partial *"pulls out" the most promising concrete state.* Partial *then creates a new abstract state S_p' that is identical to S_p, except that the two selected tuples are "marked" in S_p' by setting a tuple flag (Figure 6.c). This is to indicate that S_p' does not encode any concrete states that only include marked tuples, because those concrete states have been pulled out. The resulting concrete state and S_p' are shown in Figure 6.c. Now suppose* Partial *wants to pull out one more concrete state by refining S_p'. Then* Partial *selects t_2, the tuple with the highest score among unmarked tuples in T^Q (Figure 6.c) as the next tuple to be marked.* Partial *joins t_2 with all other marked tuples in U^Q, which is only u_1 in this case, to create concrete state $t_2 \rightarrow u_1$.* Partial *also creates a new abstract state S_p'' as shown in Figure 6.d, where t_2 has been marked.*□

In general, given an abstract state S, Rule Partial selects the most promising unmarked tuple t, joins it with all other marked tuples to create concrete states, and then creates a new abstract state where the selected tuple is marked. Note that t may not join with any other marked tuples, thereby creating no concrete state.

Rule Full is "radical" in that it *exhaustively* refines an abstract state S, generating many concrete states and incurring significant run-time costs. In contrast, Rule Partial is often "timid" in that it can pull out too few concrete states. To strike a middle ground, we develop a new rule, called Deep. Recall that when refining a state S_p using Partial, the selected tuple is joined only with marked tuples (i.e., those that have been selected before, see Example 5.2). Initially, the set of marked tuples is small, hence the joins may produce no concrete state. Consequently, Partial does not make progress, and still incurs a cost of executing the joins. This cost can be significant in our context, when we must join *across* multiple disparate databases. To address this problem, when a tuple t is selected from a tuple set, Rule Deep will join t with *all* tuples – not just the marked ones – in all other tuple sets. Deep still creates abstract states in a manner similar to Partial.

(c) Adaptively applying refinement rules: In each search iteration, once an abstract state S has been selected, Kite must decide which refinement rule, namely, Full, Partial, or Deep, should be applied to S. Kite does so in an *adaptive* fashion. Intuitively, if a rule has been applied for a while but does not lead to sufficient query processing progress, which is characterized by pruning unneeded portions of the search space, then other rules should be considered. To implement this strategy, we introduce a *goodness score* for a rule R as: $gscore(R, S) = benefit(R, S) - \alpha \cdot cost(R, S)$. The term $cost(R, S)$ represents the (estimated) cost of refining state S with rule R. Since this refinement ultimately translates into executing one or several SQL queries, we set $cost(R, S)$ to be the cost of executing these SQL queries, and estimate it using the relational query optimizers of the databases touched by the queries. The term $benefit(R, S)$ represents the relative "benefit" associated with using rule R for S. The estimation of $benefit(R, S)$ deserves some attention. Initially, all rules are assigned the same default

Domains	# DBs	Avg # tables per DB	Avg # attributes per table	Avg # approximate foreign-key joins			Avg # tuples per table	Total size
				total	across DBs	per pair		
DBLP	2	3	3	11	6	11	500K	400 M
Inventory	8	5.8	5.4	890	804	33.6	2K	50 M

Table 1. Data sets used in our experiments

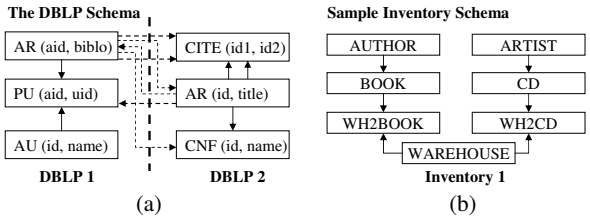

Figure 7. Schema of (a) two DBLP databases and (b) an Inventory database; cross-database FK joins are denoted with dotted lines

"benefit" for all states. As query execution progresses, Kite reduces the benefit associated with some rules and states, as follows. If a rule R has been applied to at least h states derived from an abstract state S without producing any result, then intuitively this indicates that rule R might not be good for state S, so Kite reduces $benefit(R, S)$ by a penalty factor c. In each iteration of the search, Kite then adaptively decides how to refine a state S by picking rule R^* with the highest goodness score: $R^* = argmax_R\ gscore(R, S)$.

5.3 Summary of Kite Contributions

We have described how Kite operates on an "integrated" database $\mathcal{D}$ to produce top-k answers for a query. In principle, current top-k algorithms designed for querying a single database can also be adapted to work over $\mathcal{D}$. Unfortunately, these algorithms do not scale well to multi-database scenarios. First, current CN generation algorithms often generate an unmanageable number of CNs, which makes both the CN generation and the subsequent top-k search extremely inefficient. Kite addresses this problem by lifting the level of abstraction, introducing *condensed* CNs. Second, to explore the search space encoded by the CNs, current top-k algorithms can be viewed as just applying Rules Full and Partial, both of which can lead to expensive executions in a multi-database context where distributed SQL query execution is needed. Kite addresses this problem with Rule Deep, a new exploration strategy that considers the high cost of cross-database joins. Finally, current algorithms use database statistics to decide on a refinement rule, a decision that is never revisited; this is problematic because it is often difficult to estimate statistics accurately in multi-database settings. Kite addresses this problem by adaptively selecting rules, for which Kite closely monitors their effectiveness over time.

6 Empirical Evaluation

We now describe experiments that (a) examine the run time and answer quality of Kite, (b) compare Kite with an adaptation of a state-of-the-art keyword search algorithm for a single-database scenario, and (c) measure the relative contributions of the various Kite components.

6.1 Evaluation Settings

We use two real-world data sets: DBLP consists of two databases with publication records; Inventory consists of eight databases with inventories of books, CDs, etc. (Table 1). Figure 7 show the schemas of the two databases in DBLP and the schema of a sample database in Inventory. We searched over both DBLP databases, or over two to eight Inventory databases.

We implemented Kite in Java, and ran our experiments on Oracle 10g RDBMSs over 2.8 GHz PCs with 2 GB of RAM. We implemented IR indexes with the Oracle 10g "Text Extension," and used the distributed SQL query processing facilities that Oracle provides. Similar distributed processing facilities are provided by other commercial RDBMSs (e.g., IBM DB2 and Microsoft SQL Server).

Each data point in our graphs was obtained by executing each of 10 keyword queries three times. The queries are (1) five queries whose keywords were chosen randomly from the databases and (2) five queries chosen randomly from a pool of 20 queries created by volunteers. We did not use only queries of randomly chosen keywords because we found that the chance of such keywords having any interesting association is very low (e.g., 1/20000 for two-keyword queries in an experiment), due to the large database vocabularies. Thus we asked the volunteers to create keyword queries that can possibly return meaningful associations. Query execution time is measured starting from when the query is issued until when the top-k answers have been produced, without counting offline preprocessing time, which is shared by all algorithms.

Approximate data instance matching: When applying a refinement rule, Kite executes SQL queries that frequently join tuples from different databases. As discussed in Section 3, such joins must often approximately match data instances (e.g., "M. Smith" and "Mike Smith") because of data-level heterogeneity. Many matching algorithms have been developed [6]. For the current Kite implementation, we employ the approximate string matching algorithm of [8], which exploits the query processing engines of the databases to perform matching efficiently.

6.2 Run-Time Performance

Our experiments include a baseline technique, mHybrid, which is an adaptation to our multi-database context of Hybrid, an efficient state-of-the-art top-k algorithm for keyword search over a single database [10]. Our experiments study several Kite variations, designed to identify the effect of various Kite components: Kite is the full-fledged algorithm in Section 5; k-d is Kite without Rule Deep; k-ad is Kite without Rule Deep and the ability to adaptively

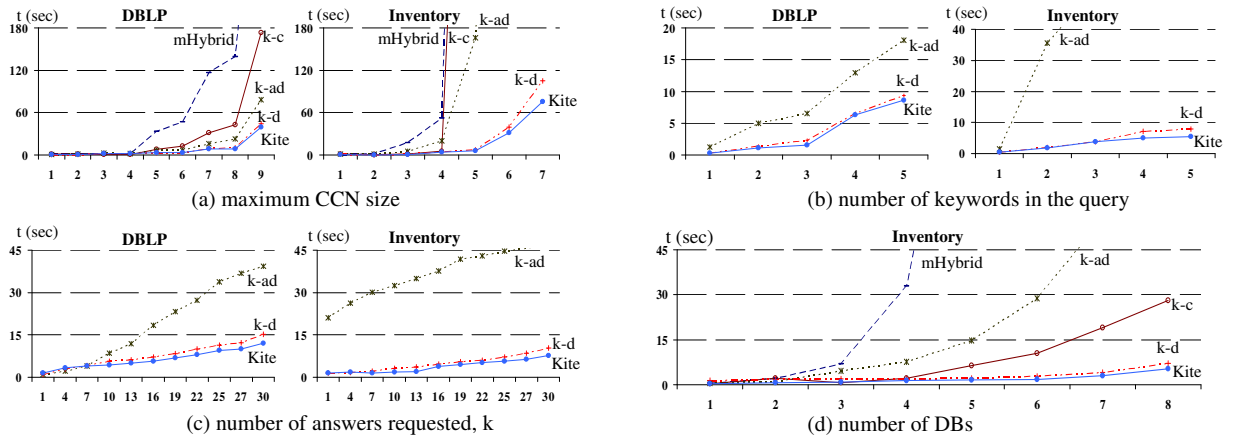

Figure 8. Run time of the Kite algorithms as a function of (a) maximum CCN size (2-keyword queries, k=10, 2 DBLP and 5 Inventory databases), (b) number of keywords in the queries (maximum CCN size = 4 in Inventory and 6 in DBLP, k = 10, 2 DBLP and 5 Inventory databases), (c) number of answers requested, k (maximum CCN size = 4 in Inventory and 6 in DBLP, 2-keyword queries, 2 DBLP and 5 Inventory databases), and (d) number of databases (maximum CCN size = 4, 2-keyword queries, k=10)

change refinement rules on-the-fly; k-c is Kite where the top-k searcher operates over CNs instead of CCNs. We examine the algorithms as we vary the maximum allowed CCN size and the number of answers requested, query keywords, and databases.

Maximum allowed CCN size: Figure 8.a plots the average run time versus the maximum allowed CCN size. The results show that mHybrid does not scale well (e.g., taking more than 180 seconds on Inventory to handle CCNs of size 5). In contrast, Kite performed well on both data sets, producing answers in reasonable amounts of time (e.g., under 6 seconds for CCNs of size 8 in DBLP and CCNs of size 5 in Inventory). Kite, k-ad, and k-d significantly outperform k-c and mHybrid, suggesting that using condensed CNs (Section 5.1) is crucial to obtain good performance. Kite also outperforms k-d, which in turn outperforms k-ad. This result demonstrates the utility of Rule Deep and of the adaptive search process.

Number of query keywords: Figure 8.b plots the average run time versus the number of keywords in the queries. Given the suboptimal performance of mHybrid and k-c, henceforth we show results for only Kite, k-ad, and k-d, for simplicity. As expected, the query length significantly affects run time. Longer queries result in larger search spaces, and in more tables touched across the databases. Our results show that Kite scales well to a moderate query size (e.g., under 10 seconds for queries of size 5). Also, Kite outperforms k-d, which in turn outperforms k-ad, demonstrating again the utility of Rule Deep and the adaptive search process.

Number of desired answers: Figure 8.c plots the average run time versus the number of answers requested, k. Kite performs well even for relatively large k values (e.g., under 15 seconds at $k = 30$ for both data sets).

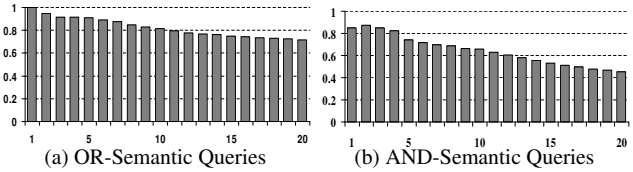

Figure 9. $P@k$ over DBLP and Inventory data sets

Number of databases: Figure 8.d plots the average run time as we vary the number of databases between one and eight in Inventory. Kite scales well up to a moderate number of databases. The algorithms with adaptive search scale much better than the non-adaptive ones: the refinement rules across databases incur a non-negligible cost of invoking the databases for SQL query execution. So rules that repeatedly fail significantly increase the run time. The adaptive algorithms detect such rules and replace them.[1]

FK join accuracy: We also measured the accuracy of the FK joins that are produced by the the join finder (Section 4). For this, we manually identified all correct FK joins across the databases and used this data to compute the precision, recall, and F_1 scores for our join finders. We found that the data-based join finder achieved 26-64% F_1, and that the schema matcher significantly improves accuracy, to 80-96% F_1. The results thus demonstrate the utility of adding schema matching to the current join discovery process.

6.3 Query Result Quality

We also assess the quality of the answers returned by Kite, compared to the hypothetical "ideal" results defined in Section 3, which involved manually integrating the multiple databases. Given a query Q, we computed its ideal result R^* as follows. First, we provided Kite with the correct

[1] We have also carried out experiments for a single-database scenario (not reported here due to space limitations) that show that Kite significantly outperforms Hybrid, the most efficient keyword search algorithm in the single-database literature [10], reducing run time by as much as 74%.

FK joins across the databases, which we identified manually. Next, we issued Q to Kite and obtained a ranked list of answers. We manually filtered this list to eliminate any spurious results originating from incorrect data-level matching of tuples. We then returned the top-20 surviving answers as the ideal result R^* for Q. This process approximates the scenario where the keyword search algorithm makes all correct join discovery and data instance matching decisions.

We then issued Q to Kite again, letting the algorithm proceed fully automatically to discover the FK joins itself and obtain a ranked list R of answers for the query. Let R_k be the top-k answers in R. For different values of k, we compute the *precision at* k of the Kite answer, $P@k$, as $P@k = \frac{|R_k \cap R^*|}{|R_k|}$, which measures the fraction of answers in R_k that also appear in the "ideal" list. Figure 9 plots $P@k$ versus k. Each data point is averaged over 20 queries (10 queries for each data set), which were selected as described in Section 6.1. We issued the queries with Boolean-AND semantics and then repeated the experiment by issuing the queries with Boolean-OR semantics. Kite managed to produce high-quality results, with high values of $P@k$ for k ranging from 1 through 20, suggesting that it can produce good approximations of the "ideal" query results.

7 Conclusions and Future Work

The problem of keyword search over multiple heterogeneous relational databases is important in many practical settings, and will become increasingly so as the number of such databases grows. We showed that a multi-database setting raises several novel challenges, and renders current single-database algorithms ineffective. To address these challenges, we introduced our Kite algorithm. Our experimental evaluation suggests that Kite scales well to multiple databases, significantly outperforms our baseline adaptation of single-database algorithms, and produces high-quality results with no need for human reconciliation of the different databases.

As future research, we will explore how to fine-tune Kite's answer scoring function (Section 3) using user feedback. For our implementation and experiments, we assigned equal weights to the three terms of this function, which capture the degree of match between queries and tuple attributes, as well as the confidence with which potentially heterogeneous attributes and data values are matched. We have conducted exploratory experiments where a human was asked to provide input on the Kite query answers by flagging incorrectly joined answers. We then used this feedback to adjust the weights of the score function, which resulted in improvements in the precision of the query answers. This anecdotal evidence leads us to believe that (moderate) human feedback can be helpful to tune the scoring function. We also plan to extend the Kite algorithm to account for communication and data-transfer costs across the databases, which should also have a positive impact on query execution efficiency, especially for widely distributed query processing scenarios.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE-02*.

[2] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient XML search with complex full-text predicates. In *SIGMOD-06*.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Authority-based keyword queries in databases using ObjectRank. In *VLDB-04*.

[4] G. Bhalotia, A. Hulgeri, C. Nakhey, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE-02*.

[5] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD-02*.

[6] A. Doan and A. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1), 2005.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS-01*.

[8] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for Web data integration. In *WWW-03*.

[9] L. Guo et al. XRANK: Ranked keyword search over XML documents. In *SIGMOD-03*.

[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB-03*.

[11] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB-02*.

[12] Y. Huhtala et al. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 1999.

[13] V. Kacholia et al. Bidirectional expansion for keyword search on graph databases. In *VLDB-05*.

[14] B. Kimelfeld and Y. Sagiv. Efficient engines for keyword proximity search. In *WebDB-05*.

[15] G. Koloniari and E. Pitoura. Peer-to-peer management of XML data: Issues and research challenges. *SIGMOD Record*, 2005.

[16] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD-06*.

[17] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *ICDE-05*.

[18] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over Web-accessible databases. *ACM Transactions on Database Systems (TODS)*, 29(2), 2004.

[19] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *ICDE-02*.

[20] S. Michel, P. Triantafillou, and G. Weikum. MINERVA$_\infty$: A scalable efficient peer-to-peer search engine. In *Middleware-05*.

[21] W. S. Ng, B. C. Ooi, and K. Tan. BestPeer: A self configurable peer-to-peer system. In *ICDE-02*.

[22] E. Rahm and P. Bernstein. On matching schemas automatically. *VLDB Journal*, 10(4), 2001.

[23] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS-05*.

[24] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *VLDB-05*.

[25] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, 1999.

[26] M. Zhong et al. An evaluation and comparison of current peer-to-peer full-text keyword search techniques. In *WebDB-05*.