Keyword Join: Realizing Keyword Search in P2P-based Database Systems

Bei Yu[1] , Ling Liu[2] , Beng Chin Ooi[3] and Kian-Lee Tan[3]
[1]Singapore-MIT Alliance
[2]Georgia Institute of Technology, [3]National University of Singapore

***Abstract*— In this paper, we present a P2P-based database sharing system that provides information sharing capabilities through keyword-based search techniques. Our system requires neither a global schema nor schema mappings between different databases, and our keyword-based search algorithms are robust in the presence of frequent changes in the content and membership of peers. To facilitate data integration, we introduce *keyword join* operator to combine partial answers containing different keywords into complete answers. We also present an efficient algorithm that optimize the keyword join operations for partial answer integration. Our experimental study on both real and synthetic datasets demonstrates the effectiveness of our algorithms, and the efficiency of the proposed query processing strategies.**

***Index Terms*— keyword join, keyword query, Peer-to-Peer, database**

I. Introduction

Keyword search is traditionally considered as the standard technique to locate information in unstructured text files. In recent years, it has become a *de facto* practice for Internet users to issue queries based on keywords whenever they need to find useful information on the Web. This is exemplified by popular Web search engines such as Google and Yahoo.

Similarly, we see an increasing interest in providing keyword search mechanisms over structured databases [6], [7], [1], [3], [2]. This is partly due to the increasing popularity of keyword search as a search interface, and partly due to the need to shield users from using formal database query languages such as SQL or having to know the exact schemas to access data. However, most of keyword search mechanisms proposed so far are designed for centralized databases. To our knowledge, there is yet no reported work that supports keyword search in a distributed database system.

We propose a keyword-join based framework that facilitates keyword search over a P2P network of autonomous databases without schema level integration. Our system capitalizes on the query processing capability of individual peers to produce potential answers or partial answers that contain some of the query keywords. Where necessary, we then integrate the information of various peers at the data level, using efficient algorithms to prune the search space. We propose a *keyword join* operator for integrating partial answers from different peers. The keyword join operates on a set of lists of partial answers (with incomplete keywords), and produces the global answers (with complete keywords) by joining partial answers from different lists based on Information Retrieval (IR) principles [10]. We also propose an efficient *keyword list* algorithm that reorganizes partial answer lists from different peers as the input lists of keyword join, so that we can generate global answers very quickly.

Our system does not pose any constraint to peers, allowing peers to remain autonomous and the network to be dynamic. In consequence, the semantics of query answering in our system is different from that of traditional data integration systems: we have a more relaxed notion of correctness and completeness of results based on the traditional IR concepts. We answer keyword queries by providing a list of potentially relevant tuples ranked with relevance scores as final answers for users to select.

The rest of the paper is structured as follows: Section II gives the framework of our proposed system. Section III describes the partial answer integration strategy in detail, which we evaluate experimentally in Section IV. Section V concludes the paper.

II. The Design Framework

In this section, we present a framework to support keyword search in a P2P environment.

A. Overview

The objective of our design is to enable peers to share their relational databases in the P2P community. A unique characteristic of our framework is to enable peers to search data using keywords, instead of issuing complex SQL queries. Given a set of keywords, our system employs IR principles to search for tuples that contain these keywords in the databases of the whole network. Each database that contains the matching tuples returns a set of potential answers or partial answers to the given keyword query. The peer that issues the query will perform the keyword join operations to integrate partial answers from different peers to generate the "best" answers where necessary. Consider an example of two databases, DB1 and DB2, residing in two different peers P_1 and P_2, shown in Figure 1. Suppose an user issues

Bei Yu is in Computer Science Program, Singapore-MIT Alliance. Tel: 68744774. E-mail: yubei@comp.nus.edu.sg.
Ling Liu is in College of Department of Computer Science, Georgia Institute of Technology. E-mail: lingliu@cc.gatech.edu.
Beng Chin Ooi is in Department of Computer Science, National University of Singapore. E-mail: ooibc@comp.nus.edu.sg.
Kian-Lee Tan is in Department of Computer Science, National University of Singapore. E-mail: tankl@comp.nus.edu.sg.

a keyword query "Titanic, 1997, DVD". Our system is able to obtain the following partial answers: from P_1, we have one partial answer tuple t_1^1 = `(Titanic, 1997, Love Story, 6.9/10)` that contains the keywords "Titanic" and "1997"; and from P_2, we have two partial answer tuples t_1^2 = `(Titanic, Paramount Studio, DVD, $22.49)` and t_2^2 = `(Titanic(A&E Document), Image Entertainment, DVD, $33.91)` both containing the keywords "Titanic" and "DVD". Now, we can join the two sets of partial answers based on certain keyword join criteria, such as substring matching between a pair of tuples. For example, t_1^1 and t_1^2 can be combined based on their common column "`Titanic`" to get the final integrated answer `(Titanic, 1997, Love Story, 6.9/10, Titanic, Paramount Studio, DVD, $22.49)`. Similarly, t_1^1 and t_2^2 can be combined. Thus, we have two complete answers to the keyword query in this example.

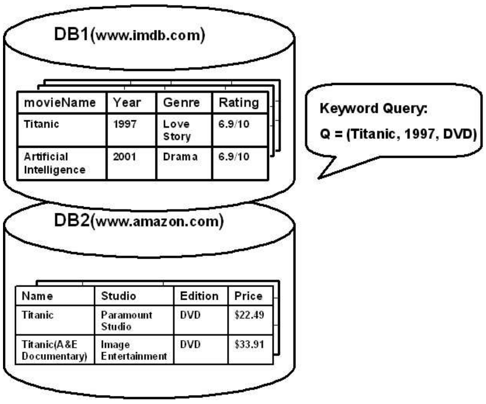


Fig. 1. An example.

B. The keyword query model

The first basic concept used in our keyword query model is *tuple tree*, which was introduced in [7], We define the tuple tree concept in our keyword query model using a somewhat more relaxed definition than that in [7].

DEFINITION 1 *A tuple tree is a tree of tuples where for each pair of adjacent tuples t_i and t_j, there are semantically correspondent column values between them, which we call* semantic links.

Note that the tree structure of a tuple tree only shows the relationship between its component tuples, we can just represent them flatly as a list of tuples for internal processing.

Our P2P database system is logically modeled as a set of relational databases $\{DB_1, DB_2, \cdots, DB_m\}$, which have heterogeneous schemas and data. A keyword query with n keywords is denoted as $Q = \{w_1, w_2, \cdots, w_n\}$. Now we define *local answers* and *global answers* to a keyword query $Q = \{w_1, w_2, \cdots, w_n\}$, in our system.

DEFINITION 2 *A local answer to a keyword query Q is a tuple tree satisfying the following three conditions: (1) The tuple tree contains at least one of the keywords in Q; (2) The tuples of the tuple tree are all retrieved from a single database, DB_i $(1 \leq i \leq m)$; (3) The tuple tree must be minimal $-$ if we remove any tuple from the tuple tree, the resultant tuple tree will contain fewer number of keywords.*

A local answer is a tuple tree provided by any peer in the system by processing the query Q over its local database. Naturally, the semantic link between tuples in a local answer should be the foreign key relationship between the tables that contain the tuples [7]. Local answers can be further classified into *local partial answers* $-$ having incomplete keywords of Q, and *local complete answers* $-$ having complete keywords of Q. In the rest of the paper we refer to the local partial answer simply as *partial answer* when there is no confusion.

DEFINITION 3 *A global answer to a keyword query Q is an* integrated tuple tree *joined by a number of local partial answers and satisfying the following three conditions: (1) Having all the keywords of Q; (2) Be minimal, i.e., if we remove any partial answer from the integrated tuple tree, the result is not a valid global answer; (3) Do not contain any two partial answers from the same peer.*

Note that the condition (3) ensures that no duplicate local answers are included in a global answer. This duplicate removal condition is especially important for unstructured P2P systems.

Given a keyword query Q, our system will return a list of desired answers that are either local complete answers or global answers. A global answer must be meaningful to users, which depends on how the local partial answers are joined. One of the technical contribution of this paper is to study how to find meaningful global answers with local partial answers from different peers. One way to approach this objective is to introduce ranking measure such that each query will be returned with a ranked list of answers, ranking in descending order according to their relevance to the query. The ranking function will be discussed in latter sections.

C. Query processing strategy

The keyword query processing in our system consists of three steps: (1) query distribution, (2) local processing of keyword queries submitted to each peer database, and (3) results merging and integration of local partial answers when necessary. We describe each of the three steps in the following subsections respectively.

1) Query distribution: When a peer, say P_a, receives a keyword query Q, it will not only search the answers in its local database, but also send the query to other peers that could have relevant information with the query keywords.

We propose to use some search mechanism to infer an *approximate relevant peer set* (ARPS) by representing the content of each local database as a "document" through transferring its local index into a dictionary as the summary of the database. In consequence, it leads to the problem of content-based text information retrieval in P2P networks. Although this problem is important and complex by itself, it is not the focus of this paper. Further, it is possible to employ some existing work on this topic, such as [4], [9], in our system.

After receiving a keyword query Q from users, P_a first finds the approximate relevant peer set (ARPS) to the query. There is a system parameter N to limit the cardinality of ARPS. Subsequently, P_a forwards the keyword query Q to all the peers in ARPS.

2) Local query processing: All the peers in ARPS will receive the keyword query sent by P_a, and perform the keyword search over their local databases as in [6].

When a peer receives a keyword query Q, it first creates a set of *tuple sets* based on the keywords in the query using its local indexes. Each tuple set R^Q is a relation extracted from one relation R of the peer's database, containing tuples having keywords of Q, i.e., $R^Q = \{t | t \in R \wedge Score(t, Q) > 0\}$, where $Score(t, Q)$ is the measure of the relevance of the tuple t in relation R with respect to the keywords in Q, which is computed with the peer's local index according to standard IR definition [6]. Next, with these tuple sets, the system generates a set of Candidate Networks (CNs), which are join expressions capable of creating potential answers. The CNs involve tuple sets and base relations (relations that do not have keywords), and they are generated based on the foreign key relationship between relations. By evaluating these generated CNs, the peer can finally produce its local answers $-$ trees of joining tuples from various relations. Each tuple tree T is associated with a score indicating its degree of relevance to the query, which is calculated as

$$Score(T, Q) = \frac{\sum_{t \in T} Score(t, Q)}{size(T)}, \quad (1)$$

where $size(T)$ is the number of tuples in T.

Having obtained its local answers to Q, a peer in ARPS separates local partial answers from local complete answers (it is possible that some peers may not have local complete answers at all). Local complete answers are ready to be output to users, since they already contain all the keywords in Q. On the other hand, the local partial answers have to be further integrated with the partial answers of other peers if possible. Each peer will return both local complete answers and local partial answers to the query initiator, P_a.

3) Results merging and integration: After P_a receives the local answers from all the peers in the ARPS, it begins to prepare output results to users. We use a user-provided parameter k_g to indicate the needed number of results. P_a first merges local complete answers from different peers, sorting them decreasingly according to their relevance scores. These results are ready to be output to the user. If the total number of local complete answers is less than k_g, P_a will integrate those local partial answers from different peers, and try to output the required number (the difference between k_g and the total number of local complete answers), denoted as K, of global answers $-$ meaningfully integrated tuple trees, to the user. Obviously, the integration part is a challenge as we do not assume any global schema information. We present our global integration solution in detail in the next section.

III. Integration of Partial Answers

In this section, we discuss the strategy for integrating local partial answers from different peers. We first analyze the requirements and then formalize the problems for integrating partial answers, and present the solutions.

A. Problem definition

Recall that there are at most N peers in the ARPS, so we have N lists of local partial answers $(L_1, L_2, ..., L_N)$. For simplicity of presentation, we assume that all lists have equal number of partial answers, l. That is, each list L_i ($1 \le i \le N$) consists of l pairs of tuple trees and the associated relevance scores to the query, $((t_1^i, s_1^i), (t_2^i, s_2^i), ..., (t_l^i, s_l^i))$, sorted in descending order according to the scores. These tuple trees all have incomplete keywords of the query, i.e., $\forall t_j^i \in L_i, keywords(t_j^i) \subset Q$. Given a query $Q = (w_1, w_2, ..., w_n)$, with n keywords, we want to integrate the local partial answers from various peers and output top K integrated tuple trees as global answers, where K is the number of integrated tuple trees the user needs.

To rank the global answers, we need to associate each global answer with a score indicating its relevance to the keyword query. We define the score of an integrated tuple tree IT as

$$score(IT) = \frac{\sum_{t \in IT} score(t)}{size(IT)}, \quad (2)$$

where t represents every local partial answer that constitutes IT, $size(IT)$ is the number of sources (peers) IT involves. This is based on the intuition that if an integrated tuple tree involves fewer sources, and has higher total score, it would be more relevant to the query since it is expected to incur less "noises" during the integration. In the example of Figure 1, the score of the integrated tuple tree t_1^1–t_1^2 is 1.585, and the score of t_1^1–t_2^2 is 1.545. We can observe that t_1^1–t_2^2 is less relevant than t_1^1–t_1^2 to the query.

In order to generate top K global answers from the set of input lists, we need to address the following three problems.

(1) Selection and organization of combinations of lists of partial answers. Considering that a global answer could potentially be joined by partial answers from any combination of input lists, naively all the combinations should be explored. However, the number of combinations increases dramatically when the number of input lists increases, which may degrade the system's efficiency greatly. The first problem is how to explore the minimal number combinations of input lists from $(L_1, L_2, ..., L_N)$ without losing possible global answers.

(2) Joining of a combination of lists of partial answers. Given a combination of lists of partial answers, the second problem is how to join them to obtain the set of integrated tuple trees as global answers, which is referred to as *keyword join.*

DEFINITION 4 *Given a keyword query Q, a set of lists of tuple trees ($L_1, L_2, \cdots, L_p$), together with a similarity threshold T, the keyword join $L_1 \bowtie_k L_2 \bowtie_k \cdots \bowtie_k L_p$ returns all set of tuple trees ($t_1, t_2, \cdots, t_p$) such that (1) $t_1 \in L_1$, $t_2 \in L_2$, $\cdots$, $t_p \in L_p$, (2) $\bigcup_{k=1}^{p} keywords(t_k) = Q$, (3) for $i \leftarrow 1$ to p,*

$\bigcup_{k=1 \wedge k \neq i}^{p} keywords(t_k) \subset Q$, *(4)* $t_1, t_2, \cdots, t_p$ *are connected into an integrated tuple tree such that for each pair of adjacent tuple trees* t_i *and* t_j, $similarity(t_i, t_j) \geq T$.

In the definition, $similarity(t_i, t_j)$ defines the information overlapping between t_i and t_j.

Observe that this problem is similar to the traditional join operation on multiple relations. However, our problem is much more complex for a number of reasons. First, in a relational table, all the tuples are homogeneous, while in a input list L_i in our problem, the tuple trees are heterogeneous because they may be generated from different CNs. There is no standard criteria to join the tuple trees. Second, when performing ordinary join operations, the join condition is determined, and values from corresponding columns of the input tables are evaluated. But in our problem, given two tuple trees, we need to decide whether there are semantically corresponding columns between them. Further, tuple trees from different databases may have conflicts in their representations of data values. Therefore some similarity heuristics are essential. Third, since our goal is to generate minimal integrated tuple trees having all of keywords in Q, we need to take care of the occurrences of keywords in the tuple trees. Finally, different from ordinary join operator which is dyadic, our keyword join need to operate on multiple input lists, and it is not associative.

THEOREM 1 *Keyword join is not associative, i.e.,*

$$(L_1 \bowtie_k L_2) \bowtie_k L_3 \neq L_1 \bowtie_k (L_2 \bowtie_k L_3).$$

The first two differences between our keyword join and traditional join operation lead to the third problem below.

(3) Similarity measure for heterogeneous tuple trees. When we perform keyword join, we need to decide whether two heterogenous tuple trees t_1 and t_2 are joinable to render meaningful information to the user. Since we do not have schema level information, the solution can only be heuristic based.

In the subsequent subsections, we will present our solutions to these three problems in reverse order for ease of explanation.

B. Similarity measure

In this section we consider the problem of how to decide whether two heterogeneous tuple trees from different peers are joinable, i.e., whether they have overlapping information. If we view each tuple tree as a flat list of column values, and we can find semantically equivalent columns from the two tuple trees respectively, they could be joined based on the common column values to render meaningful information. By "semantically equivalent", we mean that the two column values refer to the same object in real world. Basically, this problem can only be solved by heuristics since we do not have complete domain knowledge about all the databases. To this end, we make use of IR techniques to measure the similarity between two column values by treating them as "documents" [10]. Specifically, given two column values c^1 and c^2, their similarity can be evaluated with

$$sim(c^1, c^2) = \frac{\sum_{w \in C(c^1, c^2)} tf_w^1 \cdot tf_w^2}{\sqrt{\sum_{w \in c^1} (tf_w^1)^2 \cdot \sum_{w \in c^2} (tf_w^2)^2}}, \quad (3)$$

where $C(c^1, c^2)$ is the set of common tokens of c^1 and c^2, and tf_w^1 and tf_w^2 are frequencies of w in c^1 and c^2 respectively. The tokens of a column value could be space-delimited words, or q-grams – all substrings of q consecutive characters in a column value. Obviously, $sim(c^1, c^2)$ is in the range of 0 to 1, and it is 0 if c^1 and c^2 are totally different, 1 if c^1 and c^2 are exactly the same.

However it is computationally intensive to compare every pair of columns from the two given tuple trees respectively, and it will also produce many false positives because of the representation heterogeneities between different databases. For example, the number 2 could be the ID value of a student in one tuple tree, and be the serial number of a product, in another. If we join the two tuple trees based on their common column value 2, it does not make any sense.

To alleviate such a problem, we identify some "significant" column values from a tuple tree, i.e., the column values that are self-describing, and we only compare significant columns from two tuple trees respectively. For example, for a column value like "SIGMOD Conference", it is much less likely to cause ambiguity. We therefore measure the similarity between two columns as

$$sim'(c^1, c^2) = \begin{cases} sim(c^1, c^2) & \text{if } c^1,\ c^2 \text{ are significant columns} \\ 0 & \text{else.} \end{cases} \quad (4)$$

As to the task of identifying "significant" columns, it is database specific. It can be achieved through the DBAs of the local databases or by some meta-data information such as index attributes, and this information is augmented to the local answers generated by the peers. In our implementation, we select columns that are indexed by the local index as "significant" columns.

Now given two tuple trees T_1 and T_2, we can measure their information overlapping by comparing all the pairs of significant columns between them. We set the similarity score between T_1 and T_2 as the maximum among all the similarity scores of the significant column pairs. Formally, it is represented as

$$similarity(T_1, T_2) = \max_{1<i\leq l_1, 1<j\leq l_2} sim'(c_i^1, c_j^2), \quad (5)$$

where l_1 and l_2 are the number of columns of T_1 and T_2 respectively. In the example of Figure 1, $similarity(t_1^1, t_1^2) = 1$, since they have common column value "Titanic", and $similarity(t_1^1, t_2^2) = 0.5$ based on their similar columns "Titanic" and "Titanic(A&E Documentary)".

We define a threshold T such that if the similarity score of two tuple trees T_1 and T_2 exceeds T, they are considered joinable. However, the meaningfulness of the results combined from partial answers still very much depends on human observation and application-dependent preferences.

C. Top-K processing for keyword join

We use keyword joins combined with top K global answers to address the problem of how to generate top K results efficiently when performing keyword join to a given set of lists of partial answers. Note that when we perform keyword join operation on more than 3 input lists, it is difficult to use traditional query evaluation plans, such as left deep tree, right deep tree, etc., since keyword join is not associative. Given a number of tuple tree lists, we can only join them by examining every combination of tuple trees extracted from the input lists respectively. Figure 2 shows the algorithm to integrate a combination of tuple trees $t_1, t_2, \cdots, t_p$ given query Q and similarity threshold T. Lines 1 to 5 are for checking the keywords in the combination of tuple trees, and lines 7 to 13 are for checking the "connectivity" of $t_1, t_2, \cdots, t_p$.

$integrate(Q, T, t_1, t_2, \cdots, t_p)$
1. **if** the union of the sets of keywords of $t_1, t_2, \cdots, t_p$ is a subset of Q
2.   **return null**
3. **for** $i \leftarrow 1$ to p
4.   **if** the union of the sets of keywords of $t_1, t_2, \cdots, t_{i-1}, t_{i+1}, \cdots, t_p$, equals Q
5.   **return null**
6. Create two empty lists $L_{connected}$, and $L_{unCompared}$
7. Put t_1 into $L_{connected}$, put all the others into $L_{unCompared}$
8. **while** $L_{unCompared}$ is not empty
9.   **if** there are two tuple trees t and t' from $L_{unCompared}$ and $L_{connected}$ respectively, such that $similarity(t, t') \geq T$
10.   Remove t from $L_{unCompared}$
11.   Put t into $L_{connected}$
12.   Put t into $adj(t')$ $\triangleright$ adjacent list of t'
13.   **else**
14.   **return null**
15. Combine the tuple trees in $L_{connected}$ into an integrated tuple tree IT
16. **return** IT

Fig. 2. Integrate a combination of tuple trees.

To perform top-K processing for keyword join, we make use of ripple join [5], which is a family of join algorithms for online processing of multi-table aggregation queries. In the simplest version of the two-table ripple join, one tuple is retrieved from each table at each step, and the new tuples are joined with all the previously-seen tuples and with each other.

In our context, the input tables are lists of tuple trees, and we can order the tuple trees in each list in descending order of their relevance scores. The combination score of the joined result is calculated based on Equation 2, which is monotonic. Therefore, in the spanning space during joining a set of tuple tree lists, the score of the combination of tuple trees at each point is less than that of the combinations of tuple trees previously seen along each dimension. This property enables us to prune the spanning space for generating top K join results efficiently.

The pruning process works as follows. In each step, when we retrieve one tuple tree from each input list, we join each new tuple tree with previously-seen tuple trees in all other lists, in the sequence of the original order of the tuple trees in their lists, i.e., in descending order of their relevance scores. In other words, the examination of the combinations of tuple trees is towards the decreasing direction along each dimension. For example, in Figure 3, which is at step 3 of ripple join

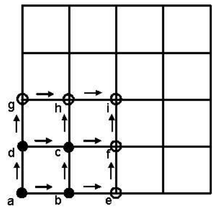


Fig. 3. An example of pruning spanning space.

between two input lists, the next sequence of combinations of tuple trees for examination would be $< e, f, g, h, i >$. Therefore, before we examine the validity of each combination of tuples trees at a point, we first calculate its combination score assuming the combination is valid. At the same time, a list L_{ans} is used to store the top K join results we currently have. We then compare the virtual combination score with the K-th largest score in L_{ans}, and if the former is smaller, we can prune it and all the rest points along that dimension. For example, as in Figure 3, suppose we are going to examine the validity of point g. We first calculate its combination score, if the score is smaller than the current K-th largest score we already have, we can safely prune the remaining points along that dimension, i.e., points $< h, i >$, since their scores must be smaller than that of point g.

In addition, if in a step all the points along all dimensions are pruned – meaning that the points in the rest of the space that have not been spanned all have smaller scores than the current K-th largest score – the algorithm could be stopped. For instance, in Figure 3, if the scores of points e and g are both smaller than the current K-th largest score, all the points in this step are pruned, and consequently we can stop the algorithm and return the current top K results.

$keywordJoin(K, Q, T, L_1, L_2, \cdots, L_p)$
1. Set p pointers $pt_1, pt_2, \cdots, pt_p$, pointing to the top unscanned tuple trees of $L_1, L_2, \cdots, L_p$, respectively
2. Set S_{low} as the K-th lowest score of the joined results obtained so far
3. **while** there is unscanned tuple tree in $L_1, L_2, \cdots, L_p$
4.   Set boolean variable $allPruned \leftarrow$ **true**
5.   **for** $i \leftarrow 1$ to p
6.   Get next tuple tree $L_i[pt_i]$ from L_i
7.   **if** $score(L_1[1], \cdots, L_i[pt_i], \cdots, L_p[1]) \leq S_{low}$ $\triangleright$ all points along i dimension are pruned
8.   **go to** 5
9. $allPruned \leftarrow$ **false**
10.   Set variables $id_1, \cdots, id_{i-1}, id_{i+1}, \cdots, id_p$ to 1
11.   **for** $k \leftarrow 1$ to p and $k \neq i$
12.   **for** $id_j \leftarrow 1$ to $pt_j - 1$ and $j \leftarrow 1$ to p and $j \neq i, k$
13.   **for** $id_k \leftarrow 1$ to $pt_k - 1$
14.   **if** $score((L_1[id_1], \cdots, L_k[id_k], \cdots, L_i[pt_i], \cdots, L_p[id_p])) \leq S_{low}$ $\triangleright$ rest points are pruned
15.   **go to** 12
16. $IT = integrate(Q, T, L_1[id_1], \cdots, L_k[id_k], \cdots, L_i[pt_i], \cdots, L_p[id_p])$
17.   **if** $IT \neq$ **null**
18.   Put IT into L_{ans}
19.   Update S_{low}
20.   Increase pt_i
21.   **if** $allPruned =$ **true**
22.   **return** L_{ans}
23. **return** L_{ans}

Fig. 4. Keyword join algorithm.

The above pruning process can be easily extended to the

TABLE I

NUMBER OF ITERATIONS OF BASIC ALGORITHM.

	$N=4$	$N=8$	$N=12$	$N=16$
$n=2$	$6l^2$	$28l^2$	$66l^2$	$120l^2$
$n=3$	$6l^2+4l^3$	$28l^2 + 56l^3$	$66l^2 + 220l^3$	$120l^2 + 560l^3$
$n=4$	$6l^2 + 4l^3+l^4$	$28l^2 + 56l^3 + 70l^4$	$66l^2 + 220l^3 + 495l^4$	$120l^2 + 560l^3 + 1820l^4$

joining on more than 3 input lists. Figure 4 shows the keyword join algorithm to produce top K integrated tuple trees from a set of lists $L_1, L_2, \cdots, L_p$.

D. Selection of partial answer lists

Now we address the problem – how to organize the N lists of partial answers so that we can generate top K global answers with keyword join quickly.

1) The basic concept: A straightforward approach is to perform keyword join operations to every k-combination ($2 \leq k \leq n$) of the N lists ($L_1, L_2, ..., L_N$) to produce potential global answers involving k sources (peers). The number of lists in a combination, k, must be smaller than n, the number of query keywords, because if a global answer has more than n partial answers, it must not be minimal, and not a valid global answer according to Definition 3.

If we assume each partial answer list has the same number of tuple trees, l, the total number of combinations of tuple trees to be examined, i.e., the maximum number of iterations of the basic algorithm, I, is

$$I = \sum_{k=2}^{n} \binom{N}{k} l^k. \qquad (6)$$

Table I shows the values of I under different values of the number of lists N and the number of keywords n. We can see that I increases very fast with the increasing of N and n. Therefore, we expect the performance of the algorithm to degrade significantly when the number of keywords or the number of peers is large.

2) Reorganization of input lists: Observe that an important requirement for a global answer is that it must both contain all the keywords of Q and be minimal. It is redundant to compare those combinations of tuple trees that cannot satisfy this requirement. For example, given $Q = (w_1, w_2, w_3)$, tuple tree t_1 has keyword set $\{w_1\}$, tuple tree t_2 has keyword set $\{w_3\}$, and tuple tree t_3 has keyword set $\{w_1, w_2\}$. It is obvious that we should not examine the combinations (t_1, t_2) – not having all the keywords of Q, and (t_1, t_2, t_3) – not minimal. We only need to examine the combination (t_2, t_3).

We therefore propose to reorganize the input lists according to the keywords of the partial answers. We maintain one list for each subset of Q (except the empty set and full set of Q), which is used to store the tuple trees that have exactly the corresponding subset of keywords. We call each list a *tuple tree keyword list* (TTKL). In total, there are $2^n - 2$ TTKLs, where n is the number of keywords in Q. Each TTKL is represented with a n-bit vector $(b_1, b_2, \ldots, b_n)$, in which b_i corresponds to a keyword w_i in Q, and b_i is set to 1 if the tuple trees in the TTKL contains w_i.

The tuple trees from $(L_1, L_2, \cdots, L_N)$ are first assigned to corresponding TTKLs according to the keywords they have. Next, we need to find out the *valid* combinations of TTKLs.

DEFINITION 5 *A valid combination of tuple tree keyword lists (TTKLs) must be both* complete *and* minimal. *A combination of TTKLs is complete if the logical OR of all the bit vectors of its element TTKL results in a bit vector with all one. A combination is minimal if an incomplete combination of TTKLs will result if any of its element is removed.*

For example, suppose there are three keywords in a query, so we will create six TTKLs, which are represented with bit vectors in {(0, 0, 1), (0, 1, 0), ..., (1, 1, 0)} respectively. All valid combinations selected from these TTKLs are: (1) $<(1,1,0),(1,0,1)>$, (2) $<(1,1,0),(0,1,1)>$, (3) $<(1,1,0),(0,0,1)>$, (4) $<(1,0,1),(0,1,1)>$, (5) $<(1,0,0),(0,1,1)>$, and (6) $<(0,0,1),(1,0,0),(0,1,0)>$.

Obviously, the number of TTKLs in any valid combination is not greater than n. Finally we integrate the partial answers by performing keyword join on all the valid combinations of TTKLs. The only difference is that in the $integrate()$ routine shown in Figure 2, we can remove lines 1 to 5, since we do not need to check the keywords, and we only need to check if the tuple trees are from different peers. We refer to this approach of using TTKL to generate global answers as *keyword list* algorithm.

However, the number of valid combinations of TTKLs for a query with n keywords still increases very fast with the growth of n. If we use $\phi(n, k)$ to denote the number of valid k-combinations of TTKLs when the number of keywords is n, Table II shows the value of $\phi(n, k)$ when n equals $2, 3, \cdots, 6$. But considering that the number of keywords of a query is very small, and it is often not bigger than four, we can therefore use the combinations of TTKLs to generate global answers efficiently. If we assume the tuple trees are distributed uniformly to each TTKL, the maximum number of iterations for keyword list algorithm, I', is

$$I' = \sum_{k=2}^{n} \phi(n,k) (\frac{Nl}{2^n - 2})^k. \qquad (7)$$

Table III shows the values of I' with different N and n. Comparing it with Table I, we can see that the number of iterations for keyword list algorithm is much less than that of the basic algorithm. The keyword list algorithm is therefore expected to be more efficient than the basic algorithm.

Further, we consider that different combinations of TTKLs usually have different contributions to the final top K global answers. Some combinations providing answers with very high scores usually contribute more answers, while some combinations even do not contribute any. Therefore, we can attempt to find the upperbound score of the potential integrated tuple trees for each combination of TTKLs and prune useless combinations where possible.

TABLE II

NUMBER OF VALID COMBINATIONS OF TTKLS VS. NUMBER OF KEYWORDS.

$\phi(n,k)$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
$k=2$	1	6	25	90	301
$k=3$	-	1	22	305	3410
$k=4$	-	-	1	65	2540
$k=5$	-	-	-	1	171
$k=6$	-	-	-	-	1

TABLE III

NUMBER OF ITERATIONS OF KEYWORD LIST ALGORITHM.

	$N=4$	$N=8$	$N=12$	$N=16$
$n=2$	$4l^2$	$16l^2$	$36l^2$	$64l^2$
$n=3$	$2.67l^2 + 0.3l^3$	$10.67l^2 + 2.37l^3$	$24l^2 + 8l^3$	$42.67l^2 + 18.96l^3$
$n=4$	$2l^2 + 0.5l^3 + 0.0067l^4$	$8.16l^2 + 4.1l^3 + 0.1l^4$	$12.45l^2 + 13.85l^3 + 0.54l^4$	$32.65l^2 + 32.84l^3 + 1.7l^4$

We get the upperbound score S_{upper} of the potential results of a combination of TTKLs by calculating the score of the combination of the first tuple trees retrieved from the TTKLs respectively.

```
Keyword list algorithm
Create 2^n - 2 TTKLs
Assign tuple trees from all local partial answer lists to the
  corresponding TTKL
Find all valid combinations of the TTKLs, put them in a set C
for each combination c ∈ C
    Calculate its upperbound score UB
Order the combinations in C in descending order according to
  their UB, and put them into a list L_C
Create a list L_glbAns to store the global answers, with variable
  S_lowest storing the lowest score of the tuple trees in it, and len
  storing the number of results in it
for each combination c from L_C sequentially
    if UB <= S_lowest and len >= K
        Output L_glbAns
        return
    else
        if UB <= S_lowest and len < K
            Output the top len results in L_glbAns first
            L_ans = keywordJoin(K - len, Q, T, c)
            Append L_ans to L_glbAns
        else ▷ UB > S_lowest
            Get rank r of UB in L_glbAns
            Output the top r results in L_glbAns first
            L_ans = keywordJoin(K - r, Q, T, c)
            Insert L_ans into L_glbAns accordingly
Output the rest results in L_glbAns
```

Fig. 5. Keyword list algorithm.

First, we order the set of valid combinations of TTKLs in descending order of their upperbound scores. Then we perform top-K keyword join for each combination of TTKLs sequentially. A list L_{glbAns} is maintained to store the top K global answers from the combinations of TTKLs already executed, and we always maintain the lowest score, S_{lowest}, of the results in L_{glbAns}, and its total number of results, len. Each time, when a new combination of TTKLs is to be executed, we first compare its upperbound score UB with S_{lowest}. If $UB \leq S_{lowest}$ and $len \geq K$, we can prune the rest of the combinations of TTKLs because the scores of their potential results must be smaller than UB, and thus smaller than S_{lowest}. Consequently, we can stop the algorithm and output current the top K global answers in L_{glbAns}. On the other hand, if $UB \leq S_{lowest}$, but $len < K$, we only need to get top $K - len$ results from the combination, since its potential results cannot change the order of the current results in L_{glbAns}. We can output the top len results in L_{glbAns} first, which can improve our system's response time greatly. Otherwise, we can get the rank r of UB in the scores of the answers in L_{glbAns}, and we only need to get top $K - r$ answers from the combination because its potential results cannot change the top r results in L_{glbAns}, and thus we can also output the top r results. The keyword list algorithm in given in Figure 5.

THEOREM 2 *The keyword list algorithm is equivalent to the basic algorithm.*

The whole proof of the theorem is lengthy and hence the sketch is provided here. Let R and R' denote the result sets generated by the basic algorithm and the keyword list algorithm, respectively. Each global answer in R is joined by partial answers such that each of them has the keywords set equivalent to that of each corresponding TTKL of a valid combination of TTKLs. This infers $R \subseteq R'$. On the other hand, each global answer in R' is joined by partial answers from the corresponding input lists from each peer. This infers $R' \subseteq R$. To conclude, $R \equiv R'$, so the two algorithms are equivalent.

IV. EXPERIMENTS

We evaluate the effectiveness and efficiency of our keyword join based integration algorithms in this section.

A. Datasets

In this experiment, we use the amalgam dataset [8] to test the quality of the integration with our similarity measure and relevance ranking method. It consists of 4 bibliography databases with similar content developed by 4 separate students. Then we use the TPC-H synthetic database for testing the efficiency of our proposed integration algorithms.

B. Quality of the integrated tuple trees

In this experiment, we measure the precision and recall of the returned results of the keyword list algorithm. We issue 10 3-keyword queries to the system, to get the average precision and recall values.

Firstly, we collect top 30 integrated tuple trees with each local database generating 40 local partial answers, and measure the precision and recall with various values of similarity threshold. It is hard to measure the recall using the standard measurement, which is the ratio of the number of relevant results retrieved to the total number of relevant results, because it is difficult to find out all meaningful integrated tuple trees to a keyword query from the databases manually. In our experiment, we use *relative recall*, i.e., we measure recall as the ratio of the number of relevant results retrieved to the total number of the meaningful results returned by the keyword list algorithm with different similarity thresholds. Figure 6(a) plots average precision and relative recall as functions of the similarity threshold for the amalgam dataset.

Observing from the figure, we can see that the precision of the results increases with the increasing of the similarity threshold, but the recall increases at first and decreases later on. This is reasonable because when similarity threshold is high, the number of returned final global answers is often few but with high quality, which leads to high precision and low recall. When similarity threshold gets lower, more global answers with decreasing overall quality are returned, so recall becomes higher and precision gets lower. As similarity threshold gets lower and lower, the quality of the returned results degrades greatly, so both of precision and recall become very low.

Next, we fix the similarity threshold to 0.3 to test the precision and recall of collecting top 30 global answers with the number of local partial answers from each peer, l, increasing from 20 to 40. Figure 6(b) reports the result of this test. From the figure, we can see that when the number of partial answers from each peer increases, the precision decreases, but the relative recall increases. This is expected since the partial answers returned from each peer are ranked decreasingly according to their relevance scores. When l is small, the overall relevance of the partial answers are better, so they tend to produce global answers with high precision. In another hand when l is small, some "good" partial answers may also be lost, so the recall is relatively low.

Finally, we fix the similarity threshold to 0.3 to test the precision and recall when the number of global answers we collect, K, increases from 10 to 30. The number of local partial answers from each peer is set to 30 in this test. Figure 6(c) illustrates the average precision and relative recall as functions of K. It can be observed from the figure that when K increases, the average precision gets lower and lower, but the recall increases. This shows that the returned answers with high ranks are generally with better quality, which reveals that our ranking function is effective.

C. Efficiency of the partial answer integration algorithms

1) Experimental setup: We use the TPC-H data for our experiments in this section. We create 16 databases, assigning each of them to one peer. Each database contains 2-4 relations of TPC-H, which are related with each other with one or two foreign key relationships of the TPC-H schema. These databases are different from each other, but have overlapping tables. We use the set of these 16 databases as ARPS for the keyword queries we issue. We integrate the local partial answers generated by these databases with our proposed integration algorithms. The average size of each database is 10 MBytes. The databases are individually managed by MySQL RDBMS. We ran all the experiments on a PC with a 2.4GHz Pentium CPU and 768MB memory. We implemented both local and global query processing in Java, and used JDBC for database connection.

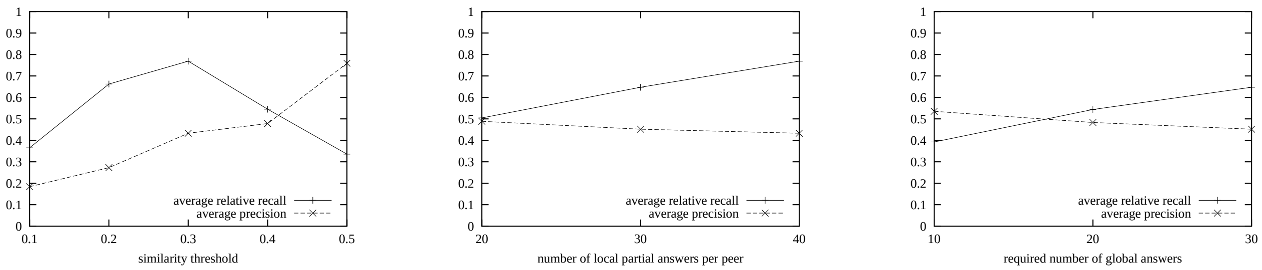
We compare the execution time of 4 integration algorithms: (1) the basic algorithm without top-K processing (BS), (2) the basic algorithm with top-K keyword join processing (BSTK), (3) the keyword list algorithm without top-K keyword join processing and any optimizations (KL), (4) the optimized keyword list algorithm with top-K keyword join processing (KLTK), in various situations. The system parameters that we vary in the experiments are (a) the number of query keywords n, (b) the required number of global answers K, (c) the number of peers in ARPS N, and (d) the number of local partial answers l generated in each peer. In all the experiments, the similarity threshold T is set to 0.3, and all the reported execution time is the average value over the execution time of 50 randomly selected keyword queries.

2) Effect of the number of query keywords: Figure 7 shows the effect of the number of query keywords to the time for integrating local partial answers by various algorithms, when the number of peers in ARPS equal to 8 and 16 respectively. In this experiment, the value of K is set to 10, the number of local answers generated by each peer l is 20. From the figures, we make the following observations:

- From the curves in the figures, we can see that the keyword list algorithm reduces the integration time dramatically compared with the basic algorithm. The KLTK is orders of magnitude faster than BS and BSTK when the number of keywords is greater than 2.
- The execution time of BS and BSTK algorithms increases much faster than that of KL and KLTK when the number of keywords increases. In Figure 7(b), when the number of keywords equals 5, both BS and BSTK can not complete the integration task within the time limit, i.e., 3000 seconds.
- The top-K processing of the keyword join are very effective, since both BSTK and KLTK are much faster than BS and KL algorithms, respectively.
- The response time of KLTK and BSTK is close to its completion time in most situations.

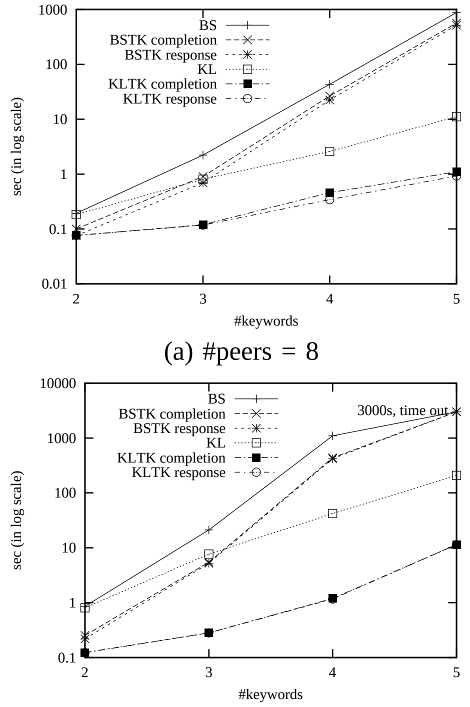
3) Effect of the number of required global answers: Figure 8 reports the effect of the number of required global answers, i.e., K, to the time for integrating local partial answers. The number of peers in the ARPS, N, was set to 16 in this experiment, and l is 20. Figures 8(a) and 8(b) show the execution time of the algorithms when the number of keywords equals 3 and 4 separately. From the figures, we make the following observations:

- The execution time of BS and KL keeps almost the same when the value of K increases from 10 to 30. This is expected because the dominating execution time of the two algorithms is used to perform the keyword join operations to the combinations of tuple tree lists, which is independent of the value of K.
- In contrast, the execution time of BSTK and KLTK increases when the value of K increases because when K is larger, it needs to take longer time to get top K results by the keyword join algorithm from each combination of tuple tree lists, and may need to perform keyword join to more combinations of tuple tree lists.
- We can also see that in both figures the response time of BSTK and KLTK increases slower than their completion time. This is because that the value of K will not affect much to the algorithms for generating first top answers that could be output before all the top K answers are generated.



(a) (b) (c)

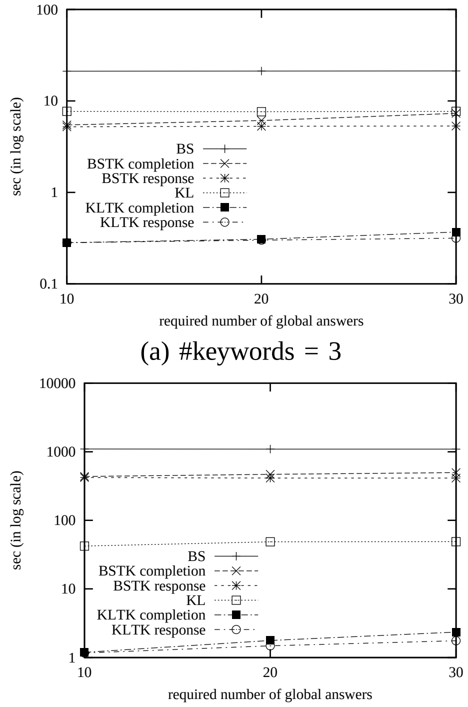
Fig. 6. The precision and recall.



(a) #peers = 8

(b) #peers = 16

Fig. 7. Effect of the number of query keywords.



(a) #keywords = 3

(b) #keywords = 4

Fig. 8. Effect of the required number of global answers.

This is desirable since when K is large, our system is still able to provide fast response to users.

V. Conclusions

We have presented a framework for keyword search in P2P-based database systems. Our system supports the keyword search interface, and enables the integration of information (partial answers) from various peers where necessary. Our proposed system avoids complex data integration, making it suitable for dynamic and ad-hoc environments and cost effective in terms of implementation. We have also proposed an efficient keyword list algorithm for generating top K global answers with our proposed keyword join operator.

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*, 2004.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[4] F. Cuenca-Acuna, C. Peery, R. Martin, and T. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *IEEE Int'l Symposium on High Performance Distributed Computing*, 2003.

[5] P. Haas and J. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, 1999.

[6] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, 2003.

[7] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, 2002.

[8] Renée J. Miller, Daniel Fisla, Mary Huang, David Kymlicka, Fei Ku, and Vivian Lee. The Amalgam Schema and Data Integration Test Suite. 2001.

[9] H. Shen, Y. Shu, and B. Yu. Efficient Semantic-based Content Search in P2P Network. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):813–826, 2004.

[10] A. Singhal. Modern Information Retrieval: A Brief Overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.