2012 IEEE 28th International Conference on Data Engineering

CI-Rank: Ranking Keyword Search Results Based on Collective Importance

Xiaohui Yu #*1, Huxia Shi *2

# *School of Computer Science and Technology, Shandong University*
Jinan, Shandong, China 250101

* *School of Information Technology, York University*
Toronto, ON, Canada M3J 1P3

1 `xhyu@yorku.ca`, 2 `huxiashi@cse.yorku.ca`

***Abstract*— Keyword search over databases, popularized by keyword search in WWW, allows ordinary users to access database information without the knowledge of structured query languages and database schemas. Most of the previous studies in this area use IR-style ranking, which fail to consider the importance of the query answers. In this paper, we propose CI-RANK, a new approach for keyword search in databases, which considers the importance of individual nodes in a query answer and the cohesiveness of the result structure in a balanced way. CI-RANK is built upon a carefully designed model call Random Walk with Message Passing that helps capture the relationships between different nodes in the query answer. We develop a branch and bound algorithm to support the efficient generation of top-k query answers. Indexing methods are also introduced to further speed up the run-time processing of queries. Extensive experiments conducted on two real data sets with a real user query log confirm the effectiveness and efficiency of CI-RANK.**

I. INTRODUCTION

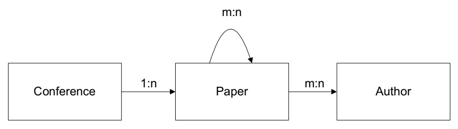
Recent years have seen a growing interest in supporting keyword search in databases [1], [4], [10], [9], [7], [12], [13]. Popularized by search engines on the Web, keyword search provides an intuitive, convenient, and effective way for users to interact with and explore the structured data stored in database. Different than the traditional ways of querying a database, it does not require the users to be familiar with the database schema and query languages.

A keyword query Q is specified by a set of keywords $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$. Since those $|Q|$ keywords could find matches in different places in the database, the results are usually interconnected structures that contain all (or as many as possible) of the given keywords. For example, one popular form of such structures is joined tuple trees [13] for keyword search in relational database systems.

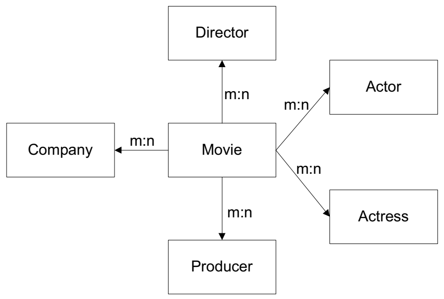
Conceptually, a database R can be viewed as a data graph $\mathbb{G}_R = (\mathbb{V}, \mathbb{E})$, where the vertex set $\mathbb{V}$ represents the data objects (e.g., tuples in the case of relational database systems), and $\mathbb{E}$ represents the connections between the objects (e.g., primary key-foreign key relationships). In general, keyword search could yield more than one result, as there could be multiple objects matching each keyword, and there could be different ways to assemble all matching tuples into a connected structure. Keyword search algorithms usually use a function $score(\cdot)$ to assign a score to a structure, and rank the set of results accordingly.

Various factors have been considered in designing the $score(\cdot)$ function. Early approaches simply use the size of the connected structure as the metric [9]. Many existing approaches utilize IR-style scoring functions [7], [12], [13]. The idea is to assign each object in the result an IR score based on its textual content, and then combine individual scores together using a score aggregation function $comb(\cdot)$ to obtain the final score. Other factors utilized in the literature include the distance between two keyword-matching objects in the connected structure [11], the coverage of keywords [13], etc.

Despite all recent efforts in improving the effectiveness of search through designing better scoring functions, one critical factor has been largely overlooked: the *importance* of the result structures. As an example, consider the following keyword query, "`Papakonstantinou Ullman`", on a bibliography database (DBLP) with the relational schema shown in Fig. 1(a).



(a) DBLP



(b) IMDB

Fig. 1. The Schemas of the DBLP and IMDB Data

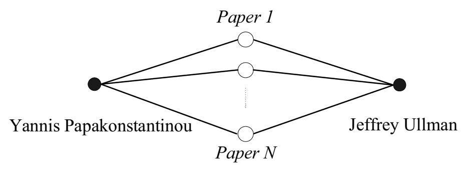


Fig. 2. Search results for the query "`Papakonstantinou Ullman`"

The two keywords in the query match two tuples in the author table containing "Yannis Papakonstantinou" and "Jeffery Ullman" respectively. Each paper co-authored by them could connect the two tuples to form a complete result (Fig. 2). Hence there could potentially exist many results, which only differ in the paper tuple that connects the two authors together. When ranking the (potentially many) results generated, it is certainly desirable to assign higher scores to the results containing papers with more citations (thus considered more important). However, existing approaches do not take this importance factor into account. The widely used IR-style scoring functions only consider the textual information contained in the results, and thus cannot distinguish the more important results from the others. Since all results have the same size, incorporating the size factor in ranking the results does not help either.

The importance of search results has been considered in ObjectRank [3], [8]. However, the goal of that work is to rank individual objects (e.g., tuples) according to their relevance to the query. This in contrast to our task where we would like to measure the *collective importance* of the objects contained in a result. ObjectRank cannot be easily extended to handle this new task. BANKS [4] and Bi-directional Search [10] only consider the importance of the root node and the nodes containing the keywords in a result tree, and do not consider the importance of intermediate nodes which we will show in Section II has a significant impact on the quality of results. Measuring the importance of objects (such as Web documents) has also been well studied in the Web area, as exemplified by PageRank [5]. However, measuring the importance of keyword search results is a significantly different task and presents unique challenges. Whereas Web documents are single objects, keyword search results are usually connected structures. In addition, Web documents are entities that already exist and their importance values can be computed offline, while for keyword search in databases, the results are obtained dynamically based on the keywords and thus their importance values have to be computed on the fly.

In this paper, we propose a novel method called CI-RANK (for Collective Importance Ranking) to rank keyword search results based on their collective importance. Note that the collective importance of a search result is not simply an aggregate of the importance of individual nodes, it is also affected by the structure of the result (i.e., how those nodes are connected together). CI-RANK takes into consideration not only the importance of the objects in the result, but also the structure of the result. To the best of our knowledge, this is the first work that addresses the issue of evaluating the overall importance of a search result. For ease of presentation, we focus on relational databases where the search results are joined tuple trees, but our approach is general enough to be applied to other types of structured data that can be modeled as graphs, such as XML data.

The underpinning of CI-RANK is a model called RWMP (for Random Walk with Message Passing). Our proposal of RWMP is inspired by the random walk model, which has been successfully employed in $\mathbb{G}_D$ [8] as well as in PageRank to compute the importance of individual nodes. The main innovation of RWMP is that it incorporates a carefully designed message passing procedure that helps model the degree of cohesiveness in the result tree in addition to the importance of the individual nodes.

CI-RANK defines a new scoring function based on RWMP. This new function calls for an algorithm that can produce the search results efficiently. To this end, we proposed a branch-and-bound algorithm that utilizes the properties of the scoring function to help prioritize the search and prune the search space. An index structure is also proposed to speedup the run-time computation through pre-computation and materialization of partial results.

The main contributions of this paper can be summarized as follows.

- We propose CI-RANK, a novel method for ranking keyword search results, which considers both the importance of nodes and the cohesiveness of the result tree structure. CI-RANK is enabled by our proposal of RWMP, a random walk model with message passing.
- We present a branch-and-bound tree search algorithm for computing the keyword search results, which allows for effectiveness pruning of the search space, while at the same time guarantees the optimality of the results.
- We design an index structure to further improve the search efficiency, and utilize the characteristics of the database graph to reduce the size of the index effectively.
- We perform extensive experiments using both synthetic and real data and queries, including manually labeled AOL query logs, that demonstrate the superiority of CI-RANK over existing IR-style methods.

The rest of the paper is organized as follows. Section II defines the problem and analyzes problems with existing ranking methods. Section III presents CI-RANK and the random walk model with message passing. Section IV describes the branch and bound algorithm for generating top-k query answers. The indexing techniques are presented in Section V. Experimental results are presented in Section VI. Section VII discusses related work. Section VIII concludes this paper and discusses possible directions for future work.

II. PRELIMINARIES

In this section, we first formally define the keyword search problem, and then provide an analysis of the problems with existing methods for ranking search results.

A. Problem Definition

A database is modeled as a weighted directed graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$. Each tuple t_i is represented by a node $v_i \in \mathbb{V}$. For any two nodes v_i, v_j, there is a directed edge $\langle v_i, v_j \rangle$ (or $v_i \to v_j$) and a backward edge $\langle v_j, v_i \rangle$ (or $v_j \to v_i$) if and only if there exists a foreign key on tuple t_i that refers to the primary key tuple v_j. The reason to model a foreign key to primary key relation as two directed edges rather than one undirected edge is to reflect the difference between these two directions. For instance, if paper p_b cites paper p_a, readers of p_b are more likely to also read p_a, whereas readers of p_a are less likely to also read p_b.

In the following discussion, we assume the AND semantics between the keywords (i.e., the results should contain all keywords).

Definition 1 (input query): An input *query* Q is a set of keywords, $Q = \{k_1, k_2, ...k_{|Q|}\}$, where k_i is keyword.

Definition 2 (free and non-free nodes [9]): If a database tuple matches any keyword k_i in the query Q, it is called a *non-free tuple* w.r.t. k_i. The corresponding node in graph is called a *non-free node* w.r.t. k_i. If a tuple does not contain the query keyword k_i, it is a *free tuple* w.r.t. k_i and the corresponding node is a *free node* w.r.t. k_i. We use $E_n(k_i)$ and $E_f(k_i)$ to denote the non-free and free node sets of k_i. From their definitions, we have $E_n(k_i) \cup E_f(k_i) = \mathbb{V}$ and $E_n(k_i) \cap E_f(k_i) = \phi$. $E_n(Q)$ is the set of all non-free nodes of the query Q. All nodes that contain any keyword in Q belong to this set. Then, $E_n(Q) = E_n(k_1) \cup E_n(k_2) \cup ... \cup E_n(k_{|Q|})$. Similarly, $E_f(Q) = E_f(k_1) \cap E_f(k_2) \cap ... \cap E_f(k_{|Q|})$ denotes the free node set of the query Q. Each node in this set does not contain *any* query keyword in Q.

Definition 3 (query answer): An answer to query Q is defined as any subtree T of $\mathbb{G}_D$ such that T is reduced w.r.t. Q. That is, there exist a set of nodes in T, $\mathbb{R} = \{v_1, \ldots, v_m\}$, where v_i contains keyword k_i $(1 \leq i \leq m)$, such that the leaves of T must come from $\mathbb{R}$, $leaves(T) \subseteq \mathbb{R}$, and the root of T must be also from $\mathbb{R}$ if the root has only one child. We call such a query answer a *joined tuple tree* (JTT).

Since there could be multiple JTTs for a query Q, a core task of keyword search in databases is to perform effective ranking of the JTTs.

B. Problems with Existing Ranking Methods

1) IR-Style Ranking Functions: Most existing solutions employ IR-style ranking. For example, DISCOVER2 [7] uses the following scoring function based on TF-IDF:

$$score(T, Q) = \frac{\sum_{v \in T} score(v, Q)}{size(T)},$$

$$score(v, Q) = \sum_{k \in v \cap Q} \frac{1 + \ln(1 + \ln(tf_k(v)))}{(1 - s) + s \cdot \frac{dl_v}{avdl_v}} \cdot \ln(idf_k),$$

where $idf_k = \frac{N_{Rel(v)}+1}{df_k(Rel(v))}$, $tf_k(v)$ is the number of occurrences of keyword k in node v, dl_v denotes the length of the text in v, and $avdl_t$ is the average length of the text in the relation which the tuple corresponding to v belongs to (i.e., $Rel(v)$), $N_{Rel(v)}$ is the number of tuples in $Rel(v)$, $df_k(Rel(v))$ is the number of tuples in $Rel(v)$ containing keyword k, and s is a constant.

As another example, SPARK [13], the state-of-the-art IR-style method, considers three factors, $score_a(T, Q)$, $score_b(T, Q)$, and $score_c(T, Q)$, where $score_a(T, Q)$ is the TF-IDF score, $score_b(T, Q)$ is called the completeness factor, and $score_c(T, Q)$ the size normalization factor. The final scoring function is defined as

$$score(T, Q) = score_a(T, Q) \cdot score_b(T, Q) \cdot score_c(T, Q)$$

The IR score of a JTT T is determined by

$$score_a(T, Q) = \sum_{k \in T \cap Q} \frac{1 + \ln(1 + \ln(tf_k(T)))}{(1 - s) + s \cdot \frac{dl_T}{avdl_{CN^*(T)}}} \cdot \ln(idf_k),$$

where $tf_k(T) = \sum_{v \in T} tf_k(v)$, $idf_k = \frac{N_{CN^*(T)}+1}{df_k(CN^*(T))}$, and $CN^*(T)$ denotes the join of relations containing the keywords (see [13] for the precise definition), dl_T is the total length of all text attributes for tuples in T, and $avdl_{CN^*(T)}$ is the average length of all tuples in $CN^*(T)$.

The completeness factor measures the coverage of keywords by T to allow flexibility in forcing AND/OR semantics, and the size normalization factor normalizes the score of T based on its size to ensure fairness between small and large JTTs. The exact formulas for those two factors are omitted here due to the limited space.

The aforementioned two scoring functions only consider the textual information, and fail to consider the difference in the importance of the nodes. Continuing the example shown in Fig. 2, we find that two paper nodes (a) "Capability Based Mediation in TSIMMIS" and (b) "The TSIMMIS Project: Integration of Heterogeneous Information Sources" can both link the two matching author nodes together into a JTT. It can be observed from the data that paper (a) is cited 7 times and paper (b) 38 times. Apparently, the JTT containing (b) should be ranked higher than that containing (a) as it is considered to be more important. However, if we use the DISCOVER2 scoring function, the two corresponding JTTs have exactly the same score, as the paper nodes do not match the keywords and therefore do not affect the final scores. Using the SPARK scoring function results does not help either: we can show that score for the JTT containing (b) is actually *lower* than that for the JTT containing (a). The reason is that the only factor that differentiates the scores of those two JTTs is dl_T with all other factors in the scoring function being equal. Since (a) has a shorter title than (b), it has a smaller dl_T and thus a higher final score.

2) Graph-based Ranking Functions: The ranking functions of BANKS [4] and Bi-directional Search [10] combine two scores, the node and edge scores, to generate the overall tree score. The node score is the average weight of the root node and the leaf nodes. The edge score is $1/(1 + sum(e))$, where $sum(e)$ is the total weight of all edges in the answer tree. However, this scoring method cannot address the difference in the importance of the free nodes. As an example, consider the

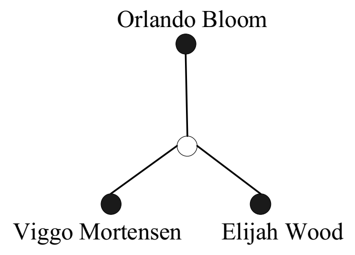


Fig. 3. The answer to the query "`Bloom Wood Mortensen`"

query "`Bloom Wood Mortensen`" on the IMDB database with the schema shown in Fig. 1(b). The result JTT may have the Actor node "Orlando Bloom" as the root, "Elijah Wood" and "Viggo Mortensen" as the leaves, and any movie co-stared by the three actors as the intermediate free node. Assume that the weights of the edges are independent of the movie. Since only the weights of the root and leaf nodes are considered, choosing which movie to link them together does not make a difference in the final score. This is certainly undesirable, because more popular movies should be favored. This example illustrates that all nodes and their connections should be accounted for in scoring the query answer.

III. Measuring the Collective Importance of Query Answers

The analysis in the preceding section reveals that the collective importance of query answers cannot be properly measured by existing ranking functions and some alternatives we have considered. This motivates our proposal of CI-Rank, which not only considers the importance of individual nodes (both free and non-free), but also the connection structures between them. Inspired by PageRank, we propose a variant of the random walk model called Random Walk with Message Passing (RWMP) to provide the basis for the design of our scoring function. In the remainder of this section, we will first describe the classic random walk model, and then present RWMP. Finally, we discuss how to score query answers with RWMP.

A. The Random Walk Model

The random walk model has been successfully applied in computing PageRank scores of Web pages [5], [14]. The whole Web can be considered as a graph $(\mathbb{V}, \mathbb{E})$, where a Web page corresponds to a node in $\mathbb{V}$, and a hyperlink is represented as an edge in $\mathbb{E}$. A random surfer moves in the graph from node to node in two ways. In any step, the random surfer currently in node v_i can either fly to a random node in the graph with probability c, or walk to a neighbor node with probability $1-c$. $c \in (0,1)$ is called the "teleportation" constant.

Let p_i denote the probability of node v_i being visited by the random surfer. It measures the importance of that node and is considered the PageRank value of the corresponding Web page. The vector $\mathbf{p} = \{p_1, p_2, ...p_i, ..., p_{|\mathbb{V}|}\}$ can be obtained by solving the following equation.

$$\mathbf{p} = (1-c) \cdot \mathbf{M}\mathbf{p} + c \cdot \mathbf{u} \tag{1}$$

The first part $(1-c) \cdot \mathbf{M}\mathbf{p}$ corresponds to a random surfer walking from a node to its neighbor, where $\mathbf{M}$ is the adjacency matrix with size $|\mathbb{V}| \times |\mathbb{V}|$. If there exists an edge $\langle v_j v_i \rangle$, m_{ij} is $1/OutDegree(v_j)$; m_{ij} is zero otherwise. The second part $c \cdot \mathbf{u}$ corresponds to the surfer flying to a random node. $\mathbf{u}$ is a probability vector called the "teleportation vector", where u_i in $\mathbf{u}$ is the probability of node v_i being the destination the surfer is flying to.

The random walk model can also be applied in the database context to compute the importance of individual nodes. Treating the database R as a directed graph $\mathbb{G}_R$, the surfer moves within $\mathbb{G}_R$ in a similar fashion as described above. The edge weights can be determined and tuned as needed, and the importance of a node is the probability that a random surfer appears in this node in a steady state. This value can be computed by iteration or Monte Carlo simulation of Equation (1).

B. Extending the Random Walk Model for Keyword Search in Databases

The random walk model works well in the Web context, but it cannot be applied directly to keyword search in databases, because the answers are trees instead of single nodes. Intuitively, we want the search results, i.e., the JTTs, to be generally important and at the same time tightly connected. Therefore, the scoring function must reflect both the importance of individual nodes and the cohesiveness of the result JTTs.

A naive way to extend the random walk model to database search is to score a JTT using the average importance values of the non-free nodes in it. The problem with this approach is that the cohesiveness of the JTT is not accounted for. For example, suppose for a keyword query $\{k_1, k_2\}$, both v_1 and v_3 match k_1, and both v_2 and v_4 match k_2. Let p_i be the importance value of v_i. If $avg(p_1, p_2) > avg(p_3, p_4)$, then the JTT containing v_1 and v_2 would be ranked higher than the JTT containing v_3 and v_4. However, if v_1 and v_2 are connected by a long path or there is no meaningful connection between the two nodes at all, while v_3 and v_4 are closely connected, the user might prefer the JTT containing v_3 and v_4.

Another possible scoring function is to take the average importance value of all nodes, free and non-free, in a JTT. This function also has a major flaw, which we call the *free node domination* problem. If the score of a JTT is dominated by the free nodes, the irrelevant JTTs that contain of irrelevant non-free nodes could be ranked higher than the relevant JTTs. For example, the answer to the query "`wilson cruz`" on the IMDB database with the schema shown in Fig. 1(b) should be a single node tree T_1 as shown in Fig. 4. However, another tree T_2, which is also shown in Fig. 4 and is obviously irrelevant, also matches this query. The non-free nodes are the movie "Charlie Wilson's War (2007)" and the actress "Penelope Cruz", respectively. They are connected by two

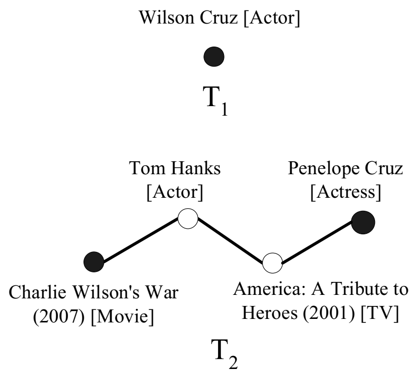

Wilson Cruz [Actor]
T1
Tom Hanks [Actor]
Penelope Cruz [Actress]
Charlie Wilson's War (2007) [Movie]
America: A Tribute to Heroes (2001) [TV]
T2

Fig. 4. The free node domination problem

free nodes, "Tom Hanks" and "America: A Tribute to Heroes (2001)". Because the importance value of the free node "Tom Hanks" in T_2 is much higher that of the single node in T_1, the average importance value of T_2 is higher than T_1, and is therefore ranked higher. This illustrates the problem caused by domination of free nodes.

Yet another alternative ranking function is to use *the average importance score / result size*, which takes not only the average importance of nodes, but the size of the result into consideration. The problem is that it cannot address the structural difference between the answer trees. For example, suppose we have two JTTs, T_1 and T_2, for the same query. T_1 has a star structure with one free node in the center and four non-free nodes connecting directly to the free node. T_2 also has one free node and four non-free nodes. They have exactly the same importance scores as those in T_1, but they form a chain-like structure with the free one in the center. Since T_1 and T_2 have the same size, and the importance values of the nodes are the same, T_1 and T_2 will have the same ranking scores despite their significant structural difference and hence potentially very different semantics.

C. Random Walk with Message Passing

We propose a random walk model with message passing (RWMP) that is designed to match the characteristics of keyword search in databases. It considers both the importance of the single nodes and the structural cohesiveness of the answer trees in a balanced way, avoiding the pitfalls of the ranking functions discussed in the preceding subsection.

1) Overview of the Model: The RWMP model consists of the following procedures.

1. Message Generation: In RWMP, a non-free node generates messages indicating the existence of itself. The messages generated from different nodes have different types. We call such a message generated at v_i a v_i message. The number of messages generated at v_i is proportional the importance value of v_i. Let p_i be the importance value of the node v_i, and $|v_i|$ the number of words in node v_i. $|v_i \cap Q|$ denotes how many words in the node v_i match the query Q. The number of the message notes generated at node v_i is $r_{ii} = (t \cdot p_i \cdot |v_i \cap Q|)/|v_i|$, where t is the total number of the random surfers in the graph.

2. Message Passing: One of the design objectives of RWMP is to measure the strength of pairwise connection of the non-free nodes in an answer tree. It is very difficult to measure the connection inside an answer tree using the original random walk model. We therefore add to it a component called message passing.

Intuitively, the message-passing model can be thought of as follows. A source node (which contains a keyword in the query) sends out a signal. The strength of the signal is weakened when it passes through an intermediate node. By measuring the strength of the received signal in the destination node, we know how closely these two nodes are connected.

To be more specific, the random surfers move to the neighboring nodes, carrying the messages. Since we are interested in evaluating the cohesiveness of the result tree, we assume that messages are only passed between nodes inside the tree and the surfers leaving the tree do not carry any messages. The number of surfers moving along different outgoing edges are proportional to the weights of those edges (which can be determined by domain experts or in ways similar to what is done in BANKS [4]).

Let the number of v_i messages received at node v_j be r_{ij}, and the number of leaving messages be f_{ij}. Suppose v_k is a neighbor of v_j and v_k is not in the path that connects v_i and v_j in the tree T. Then, at node v_k, the number of the v_i messages received, r_{ik}, is $f_{ij} \cdot w_{jk} / \sum_{v_n \in N(v_j) \cap V(T)} w_{jn}$, where w_{jk} is the weight of edge e_{jk}, $N(v_j)$ is the set of neighboring nodes of v_j in $\mathbb{G}_R$, $\sum_{v_n \in N(v_j) \cap V(T)} w_{jn}$ is the sum of the weights of all edges from v_j to its neighbors in the tree T. If the neighbor v_k is on the path that connects v_i and v_j in the tree T, v_k must have already got the v_i messages before v_j does. Since v_j sends v_i messages along all of its outgoing edges in the tree T, some v_i messages are sent back to v_k. These messages are discarded.

3. Message Dampening: The number of messages is dampened in intermediate nodes, as surfers may drop with a certain probability some of the messages they are carrying. We call $d_j = f_{ij}/r_{ij} < 1 \quad (i \neq j)$ the dampening rate at node v_j. Its value is positively correlated with the importance of node v_j, and is the same across different message types. Apparently, the higher the dampening rate is, the more messages are kept and passed on. The particular choice of this dampening rate is discussed in the sequel.

2) Methods for Dampening Messages: Message dampening is an important procedure in RWMP as the rate of dampening reflects the relative importance of intermediate nodes. Since we prefer the JTT to be connected by important nodes, the relationship between the dampening rate d_i and the node importance p_i should be monotonically increasing. A straightforward design is $d_i \propto p_i$. However, its dampening strength is too heavy. In our experience, a high importance value can be thousands of times greater than a low one. As such, the range of the dampening rates generated according to $d_i \propto p_i$ is too large and inflexible.

Our preliminary experimental results reveal that our model is highly effective when the dampening rate of an intermediate node is proportional to the logarithm of its importance value. To achieve this effect, we devise the following in-node message exchange process. Suppose that in a node v_i, there are $t \cdot p_i$ random surfers. Each random surfer carrying messages talks to all other surfers inside this node in several steps:

(1) The surfer finds a group of g random surfers (regardless of the types of messages they carry) and talks to them. With a certain probability, she gives the message to one of the surfers in this group. We call this message-carrying surfer a speaker, and the group of g random surfers listeners.

(2) The listeners in the previous step become speakers. Each of them finds g listeners and talks to them with possible message exchange. A random surfer can be a listener only once in any step.

(3) Repeat this process, until all surfers in this node have communicated directly or indirectly with the given surfer.

In this manner, the surfer can communicate with all random surfers in this node in $log_g(t \cdot p_i)$ steps.

Let the probability that a surfer is willing to keep the messages by herself be α. With probability $1-\alpha$, she gives the messages to one of her listeners. In the last talk step, if the surfer who is now carrying the messages does not want to keep the messages, she just discards them. Hence, after $log_g(t \cdot p_i)$ talk steps, the probability that a message is not discarded is $\alpha \cdot \sum_{n=0}^{log_g(t\cdot p_i)} (1-\alpha)^n = 1-(1-\alpha)^{1+log_g(t\cdot p_i)}$. This is the dampening function at this node.

The dampening function has three parameters, the probability of keeping the messages α, the group size g, and the total number of random surfers t. Recall that the node importance p_i is obtained from Equation (1). To simplify the dampening function, we assume that the node with the lowest importance value has only one random surfer. Let p_{min} denote the importance of this node. Then, $t = 1/p_{min}$, and the dampening rate is simplified as follows.

$$d_i = 1-(1-\alpha)^{1+log_g(p_i/p_{min})} \tag{2}$$

3) Scoring of JTTs: In RWMP, a non-free node v_i receives messages generated from other non-free nodes. The number of the types of the incoming messages is $|E_n(Q) \cap V(T)| - 1$. We choose to use the number of messages of the least populous type in v_i as its score, i.e.,

$$score(v_i) = \min_{v_j \in E_n(Q) \cap V(T), j \neq i} (f_{ji}) \tag{3}$$

Intuitively, in a destination node (non-free node), if we take one message of each type (corresponding to each source node) and combine them, that can be considered complete knowledge of all sources. The number of messages of the least populous type determines the number of such message combinations, which measures how strongly this node is connected to the other parts of the answer tree. The scoring function in Equation (3) reflects the number of such combinations.

The scoring function of CI-RANK for the whole tree T is calculated as the average of the scores of all non-free nodes.

$$score(T) = \frac{\sum\limits_{v_i \in E_n(Q) \cap V(T)} (score(v_i))}{|E_n(Q) \cap V(T)|} \tag{4}$$

4) Benefits of CI-RANK*:* The benefits of the scoring model in CI-RANK can be summarized as follows.

	Characteristics of the model
1	The number of the source messages is proportional to the importance of the source node. **Effect:** The important non-free nodes are favored.
2	The number of the messages is dampened when they pass through a node. **Effect:** Messages in a larger JTT have to go through more intermediate nodes before reaching other non-free nodes and thus tend to be dampened more heavily, resulting in lower scores. Smaller trees are preferred.
3	The relation between the dampening rate and the node importance is monotonic. **Effect:** Preference is given to important free nodes in connecting the non-free nodes.
4	The score of the tree is not dominated by the free nodes. **Effect:** The free node domination problem is avoided.

TABLE I

BENEFITS OF THE MODEL

IV. SEARCH ALGORITHMS

We now consider the problem of efficiently generating the top-k answers to the keyword queries. In many previous studies, the cost of an answer tree is the summation of its edge weights. In this case, the problem of generating the top-k answers becomes the minimum cost group Steiner tree problem, which has been shown to be NP-Complete. [17] lists several heuristic algorithms for this problem.

The scoring function in CI-RANK is more complex than the summation of the tree edge weights. Therefore, existing algorithms cannot be applied directly. In what follows, we will first describe a naive brute-forth algorithm, and then present a branch and bound search algorithm that utilizes the characteristics of the scoring function to get the top-k answers in a more efficient way. Similar to previous studies [4], [9], [7], we put a limit D on the diameter of answer trees.

A. The Naive Search Algorithm

In the naive search algorithm, we first perform breadth first search from all non-free nodes. Then, the information gathered during the search is combined to generate the answer trees. The maximal search distance is $\lceil D/2 \rceil$ from each non-free node. When a node is visited in the search, the source node and the distance between the current node to the source node is recorded in this node. Because there may exist several different paths from the source node to this node being visited, the node visited right before this node is also recorded. After the search is finished for all non-free nodes, we check the visited nodes. If a node is reachable from one or more non-free nodes, and all keywords in the query are covered by these reachable non-free nodes, this node is selected as the root of the answer tree. From this root node, we connect the non-free nodes together to generate a full answer. Since there could be multiple paths from a root node to each non-free node, all combinations should be considered to generate different answers.

The main problem with this naive search algorithm is that it has to thoroughly expand all non-free nodes to find the best answer, a very time consuming task. To improve the efficiency, we design a branch and bound search algorithm, which prefers to expand the most promising non-free nodes and can stop early without exhaustively expanding all non-free nodes.

B. The Branch and Bound Search Algorithm

The branch and bound search algorithm works on candidate trees. A candidate tree C is a tree that covers at least one keyword. The initial candidate trees are the single non-free nodes constrained by the query keywords. We use *tree grow* and *tree merge* procedures [6] to expand small candidate trees to larger ones. The complete tree resulting form the expansion process that matches all input keywords, is considered an answer. At any time, only the top-k answers are kept.

The algorithm orders the candidate trees according to their upper bound values and expands the candidate tree with the highest upper bound value. Let us denote a candidate tree C with a root node v_i by $C(v_i)$. The upper bound of this candidate tree $C(v_i)$, $ub(C(v_i))$, is the maximal score that an answer tree can have if it is obtained from expanding this candidate tree $C(v_i)$. All candidate trees are kept in a priority queue P in the decreasing order of their upper bound values. The algorithm always retrieves the tree at the head of the queue P, the most promising candidate tree, and expands it. When the minimal score in the the current top-k answers is higher than the upper bound value of the tree at the top of the queue, the algorithm can stop, because any answer trees generated in future steps are guaranteed to be worse than the top-k answers already obtained.

The outline of the branch and bound algorithm is shown in Algorithm 1.

Algorithm 1 Branch and bound search

```
1: Initialize the priority queue $P := \phi$
2: Initialize the list of top-$k$ results $L_{top-k} := \phi$
3: for each $v_i \in E_n(Q)$ do
4:     create a single node candidate tree $C(v_i)$
5:     enqueue $C(v_i)$ into queue $P$
6: end for
7: while $P \neq \phi$ do
8:     dequeue $P$ to $C(v_i)$
9:     if $|L_{top-k}| = k$ and $ub(C(v_i)) < minscore(L_{top-k})$ then
10:        return $L_{top-k}$
11:    end if
12:    $S_{grow}(C(v_i)) \leftarrow$ tree grow of $C(v_i)$
13:    for each $C(v_j) \in S_{grow}(C(v_i))$ do
14:        $L_{top-k} := L_{top-k} \cup \{C(v_j)\}$ if $C(v_j)$ is complete
15:        enqueue $C(v_j)$ into queue $P$
16:        $S_{merge}(C(v_j)) \leftarrow$ tree merge of $C(v_j)$
17:        for each $C(v_h) \in S_{merge}(C(v_j))$ do
18:            $L_{top-k} := L_{top-k} \cup \{C(v_h)\}$ if $C(v_h)$ is complete
19:            enqueue $C(v_h)$ into queue $P$
20:        end for
21:    end for
22:    if $|L_{top-k}| > k$ then
23:        truncate $L_{top-k}$ to size $k$
24:    end if
25: end while
```

Tree growing and tree merging: Let $v_j \in N(v_i)$ be a neighbor of v_i. If v_j is not contained in tree $C(v_i)$ (i.e., $v_j \notin C(v_i)$), we can create a new tree whose root is v_j with a single child tree $C(v_i)$. This process is called tree growing. $C(v_i)$ can merge with another candidate tree $C^{'}(v_i)$ if they have the same root, and the result covers more keywords than either $C(v_i)$ or $C^{'}(v_i)$. Invalid merge results with cycles are avoided through a sanity check.

Upper bound calculation: The tree growing and tree merging methods ensure the following important property of the candidate tree: when a candidate tree $C(v_i)$ is used as part of a larger tree $C(v_j)$, only the root of this small candidate tree, v_i, is connected to the other part of $C(v_j)$. $ub(C(v_i))$ is calculated according to this property from two different estimates, the *complete estimate* and the *potential estimate*, denoted by $ce(C(v_i))$ and $pe(C(v_i))$ respectively.

Complete estimate: The complete estimate $ce(C(v_i))$ is the highest possible score that an incomplete candidate tree $C(v_i)$ can achieve when it is expanded to a complete answer. An incomplete candidate tree $C(v_i)$ only covers a portion of the query keywords. To expand it to a complete tree, we need to supply non-free nodes that cover the missing keywords, and connect them by some paths to the root v_i of this candidate tree. If $C(v_i)$ is complete, the complete estimate is its own score $score(C(v_i))$. The difficulty of obtaining the complete estimate $ce(C(v_i))$ for an incomplete candidate tree $C(v_i)$ is that we do not know which non-free nodes can be used to complete the tree and how to choose the paths that connect them to the candidate tree. The fact that some parts of these paths can be shared makes the above decision even more difficult. To obtain a rough estimate, we assume that the best supplementary non-free nodes are connected directly to the root v_i of the candidate tree $C(v_i)$. This assumption leads to a complete tree that may not exist in reality. It is thus called a hypothetical tree.

The complete estimate given by the above method is usually much larger than the true value, and can be further refined by checking the direct neighbors of the root v_i. If a missing keyword can be covered by a node v_j that directly connects to v_i, we consider two scenarios. One is that v_j is contained in the hypothetical complete tree; the other is that this keyword is covered by a node not directly connected. In the latter case, we assume that the best supplement node is connected to the root v_i through its best neighbor. The greater estimate from these two scenarios is chosen as the complete estimate. This method provides a more accurate estimate, and is used in our experiments.

Potential estimate: Although a complete answer tree covers all input query keywords, it is still possible to add more non-free nodes to this tree to generate other complete trees with higher scores. The potential estimate, $pe(C(v_i))$, is the maximal contribution from the additional nodes appended to a complete tree. The tree growing and tree merging method ensures that the additional nodes can only connect to a candidate tree $C(v_i)$ through its root v_i. Similar to the method to get the complete estimate $ce(C(v_i))$, we can consider the non-free nodes in the set of nodes consisting of v_i and all direct neighbors of v_i. All other non-free nodes are assumed to be connected to v_i through the best neighbor of v_i. When these additional nodes are added to the complete candidate tree $C(v_i)$, each of them will have a node score, which is defined in Equation (3). We take the maximum among these node scores as the potential estimate $pe(C(v_i))$.

Combination of complete and potential estimates: The complete and potential estimates can be combined to give the upper bound for the candidate tree, $ub(C(v_i))$. The upper bound of a candidate tree $T(v_i)$ is the greater of the complete and potential estimates. This is stated in the following lemma.

Lemma *1 (Correctness of the upper bound):* For an arbitrary full answer tree T grown from the candidate tree C, $score(T) \leq ub(C) = max(ce(C), pe(C))$.

Theorem *1 (Optimality of the search algorithm):* The top-k results of the branch and bound search algorithm are guaranteed to be optimal. For any answer tree $T \notin L_{top-k}$ where L_{top-k} is the top-k results, $score(T) \leq minscore(L_{top-k})$, where $minscore(L_{top-k})$ is the minimal score of the top-k results.

Proof: Prove by contradiction. If the top-k results L_{top-k} generated by the branch and bound search algorithm are not optimal, there should exist an answer tree T such that $score(T) > minscore(L_{top-k})$. Because this tree T is not generated by the branch and bound search algorithm, at least one of the candidate tree, denoted by C, that can be grown or merged to T is pruned. The pruning rule in Lines 24 and 25 of Algorithm 1 shows that $ub(C) \leq minscore(L_{top-k})$.

On the other hand, Lemma 1 states that $ub(C) \geq score(T)$. Considering the assumption of tree T, $score(T) > minscore(L_{top-k})$, we can get a result that $ub(C) > minscore(L_{top-k})$.

It is a contradiction that $ub(C) \leq minscore(L_{top-k})$ and $ub(C) > minscore(L_{top-k})$. Therefore, the initial assumption that the top-k results L_{top-k} are not optimal must be false. ■

Further improvement: The performance of this algorithm is sometimes impaired by noisy non-free nodes. These noisy nodes cannot connect to other non-free nodes to generate complete answers, but they have very high importance values. In some cases, the branch and bound search algorithm wastes a lot of time to enumerate the candidate trees grown from these noisy nodes because they have higher upper bound estimates. To alleviate such problems and obtain more accurate upper bound estimates, we build indexes on the database graph, as discussed in the following section. The index improves the evaluation of the upper bounds of the candidate trees during their generation in Lines 12 and 16 of Algorithm 1.

V. Indexing Techniques

We design indexing methods to further improve the search efficiency of the branch and bound search algorithm.

A. Naive Indexing

In a naive indexing scheme, we could compute offline and store the shortest distances between any two nodes in the data graph, $DS(v_i, v_j)$, and the minimal loss of messages due to dampening when surfers move from one node to another, $LS(v_i, v_j)$. The shortest distance information gives more pruning power at run-time. When expanding a candidate tree, a non-free node can be ignored if it is too far away from the root of the candidate tree such that the diameter of the result tree would exceed the limit D. The information on the minimal loss of messages improves the accuracy of the upper bound estimate used in the search.

The main problem of this indexing scheme is that its space complexity is $O(|\mathbb{V}|^2)$, too big even for databases of moderate sizes. This leads us to the development of better indexing schemes. In particular, we consider the special case where the database has a star schema, which is common in OLAP applications, and introduce a new indexing method called *star indexing*. This indexing method dramatically reduces the space requirement and strikes a balance between the size of the index and its pruning power.

B. Star Indexing

In star indexing, only part of the nodes, which we call star nodes, are indexed. All nodes in the star table are considered star nodes, where a star table is defined as a table such that if it is removed, all the remaining tuples in the database become disconnected. For example, in Fig. 1(b), the Movie table in the IMDB database is a star table. Similarly, the Paper table is a star table in the DBLP database as shown in Fig. 1(a). It is possible that tuples in the database cannot be completed disconnected by the removal of a single star table. In this case, we can have multiple star tables. The nodes in all of the star tables are indexed, maintaining the same information as in naive indexing. This design results in a significant reduction of the nodes that have to be indexed.

There are several different cases in leveraging the proposed star index to speed up the branch and bound algorithm by evaluating the relationship between two nodes.

Case 1: Both nodes are star nodes. The shortest distance $DS(v_i, v_j)$ and the minimal loss of the messages $LS(v_i, v_j)$ can be retrieved directly from the index.

Case 2: A star node v_i and a non-star node v_j. In this case, we first retrieve the set of star nodes directly connected to the v_j. Let it be $S_{star}(v_j)$. Then, the shortest distances and minimal message losses are retrieved for all $v \in S_{star}(v_j)$ and v_i. We select the star node v_h from $S_{star}(v_j)$ that gives the lowest values among all retrieved shortest distances. The shortest distance between v_h and v_i is denoted by $DS(v_h, v_i)$. If v_j is on the shorted path from v_h to v_i, the shortest distance between v_i and v_j is $DS(v_h, v_i) - 1$. This is shown in Fig. 5(a). The star nodes are represented by star symbols, and the other nodes are depicted as circles. Other possible values for the

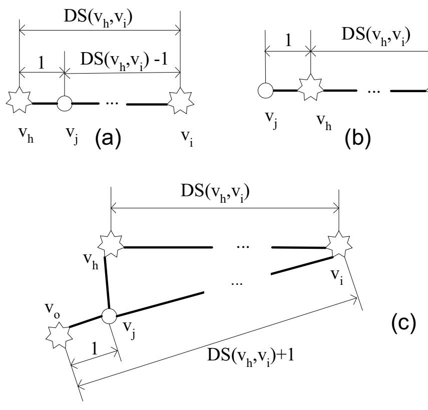


Fig. 5. The shortest distance between a star node and a non-star node

shortest distance between v_j and v_i are $DS(v_h, v_i) + 1$ and $DS(v_h, v_i)$, illustrated in Fig. 5(b) and Fig. 5(c) respectively. The latter value is given when there is another star node v_o that connects directly to v_j, which has a shorest distance $DS(v_h, v_i) + 1$ to node v_i, and has v_j on its shortest path. Therefore, we cannot get the accurate shortest distance in this case. In our search algorithm, we use the lower bound, $DS(v_h, v_i) - 1$ as the shortest distance. The pruning power is reduced because some nodes that should be pruned might be kept due to the inaccuracy thus introduced. This illustrates the trade-off between the index size and its pruning power. The minimal loss of messages is calculated in a similar way.

Case 3: Both nodes are non-star nodes. We expand both non-star nodes to two sets of star nodes and calculate the approximate shortest distance in a manner similar to the second case.

Benefits: Star indexing enhances the efficiency of the branch and bound search algorithm in two ways. First, it can provide a more accurate complete estimate. Without the index, if a supplement node is not directly connected to the root of the candidate tree, we just assume that it is connected to the best neighbor of the root. However, the supplement node may be far from the root, or even not connected at all. The minimal loss of messages between two nodes, which we can get from the index, is used to improve the accuracy of the complete estimate. Second, we can avoid computing the potential estimate and get a more accurate evaluation of the contribution from the additional nodes. The contributions of all non-free nodes that are not in the candidate tree can be calculated using the index. A greedy algorithm retrieves them in descending order of the contributions, and adds them into the candidate tree until the score of the result tree starts to decrease. Because the star index does not store the precise values for the loss of the messages, the calculated contribution is not an exact value but rather an upper bound.

VI. Experiments

A. Experiment Setting

To evaluate the effectiveness and efficiency of CI-Rank, we conduct extensive experiments on two large-scale data sets and a real user query log.

The data sets used in the experiments are the Internet Movie Database (IMDB)[1], and the DBLP data[2]. Both data sets are modeled as graphs. The IMDB data contains $3,378,743$ nodes and $28,482,926$ edges, and the DBLP data contains $2,132,821$ nodes and $8,446,804$ edges respectively. The schemas of the two databases are shown in Fig. 1.

Similar to the previous study [3], the edge weights are chosen experimentally when modeling the databases as graphs in our experiments. We find the edge weights listed in Table II work well in our experiments. These weights will be normalized in the graph. For example, one movie has three out edges, which point to an actor, a director and a producer. Before the normalization, the weights of these edges are 1.0, 1.0, and 0.5. The random walk model requires that the weights of out edges of a node sum to 1.0. Therefore, the weights of above three out edges are normalized to 0.4, 0.4, and 0.2. There is another problem in modeling the IMDB data set due to its normalization. The same person may have several copies because he/she has different roles. For example, Mel Gibson is the director of the movie Braveheart (1995). At the same time, he is an actor in it. When we map the IMDB data set to a graph, two nodes, director Mel Gibson and actor Mel Gibson, are created. As a result, the importance value belonging to the same person is split and distributed into two different nodes. To avoid this case, we merge such nodes into a single node. In the above example, we only have one Mel Gibson node in the graph. This node has two different edges, directing and acting, that point to the node of the movie Braveheart (1995).

A real user query log is used in our experiments. In 2006, AOL provided the user query log accumulated in their search engine for three months. The log, which can be obtained from a mirror site[3], contains 20 million queries issued by $650,000$ users. Each record in this log has both the query content and the URL clicked by the user. Only the records whose clicked URL contains http://www.imdb.com are collected. The total number of such records is $81,250$. Among them, we manually labeled $29,078$ frequent queries. For robustness, each of those queries must have appeared at least three times in the log. These labeled queries are used as user feedback to bias the CI-Rank model. 44 complex queries are selected from the user query log to verify the effectiveness and efficiency. These complex queries, which match at least two nodes, have clear meaning and no ambiguity in the manual labeling. Since the AOL log does not contain any queries related to DBLP, 20

[1]http://www.imdb.com/interfaces
[2]http://dblp.uni-trier.de/xml/
[3]http://www.gregsadetsky.com/aol-data/

Data set	Edge type	Weight
IMDB	Actor → Movie	1.0
	Movie → Actor	1.0
	Actress → Movie	1.0
	Movie → Actress	1.0
	Director → Movie	1.0
	Movie → Director	1.0
	Producer → Movie	0.5
	Movie → Producer	0.5
	Company → Movie	0.5
	Movie → Company	0.5
DBLP	Conference → Paper	0.5
	Paper → conference	0.5
	Author → Paper	1.0
	Paper → Author	1.0
	Citing paper → Cited paper	0.5
	Cited paper → Citing paper	0.1

TABLE II
THE EDGE WEIGHTS

synthetic queries are used instead. The method to generate the synthetic queries is explained in the following subsection.

CI-RANK is implemented in Java. The index is built using Apache Lucene. All the experiments are run on an IBM Linux server with a 3.0GHz Intel Dual Core processor, 4GB of RAM, and 2TB SATA HD RAID.

B. Effectiveness

We use two metrics, mean reciprocal rank and precision, to measure the effectiveness. The reciprocal rank is the inverse of the rank of the best answer. The best answer is decided by a user study. Five graduate students were invited to select their most preferred answer for each query. For a query, the best answer is decided by majority voting. In the case of a tie, all of the answers are considered the best. Another measure, precision, is the fraction of the answers generated that are relevant to the query. Graded relevance levels are used. If a relevant answer does not contain all input keywords, its relevance level is penalized by the percentage of the missed keywords.

We first study the effect of two parameters of CI-RANK, α and g. Both parameters defined in section III-C are used to control the message dampening process inside a node. α is the probability a message carrier is willing to take the messages in one talk. g is the clique size of the talk. The importance values of the graph nodes are calculated by the iteration of Equation (1). The typical value, 0.15, is used for the teleportation constant c in this equation.

Fig. 6 and 7 illustrate how the effectiveness is adjusted by the parameters α and g. The change of the mean reciprocal rank has same pattern in both data sets. When α is between 0.1 and 0.25, the result is better than the α out of this range. Similarly, the parameter g in a range from 10 to 20 gives relatively better accuracy.

We choose $\alpha = 0.15$ and $g = 20$ for the following experiments since CI-RANK has best performance with them for both IMDB and DBLP data sets. The value range of the dampening rate in CI-RANK is controlled by α and g. α is the

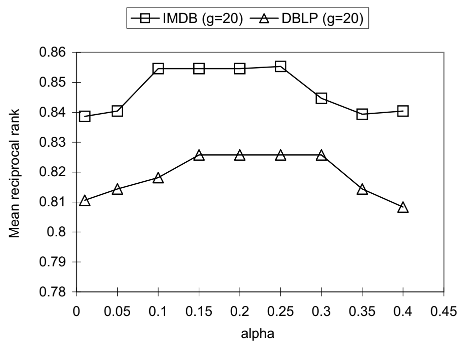


Fig. 6. The effect of α on the mean reciprocal rank

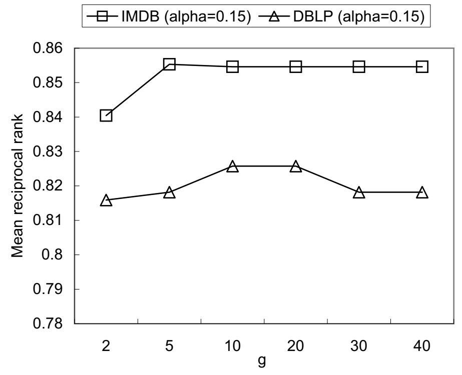


Fig. 7. The effect of g on the mean reciprocal rank

minimal value of the dampening rate. If α is fixed, the maximal value of the dampening rate decreases when g increases.

To compare CI-RANK and existing methods, we implemented SPARK's [13] scoring function on the database graph, as well as BANKS [4]. The result of CI-RANK is obtained with the default parameters $\alpha = 0.15$ and $g = 20$. No user feedback is used. Two different query sets are used for IMDB. One is obtained from the AOL user log, and the other synthetically generated. Interestingly, most of the complex queries obtained from the AOL user log can be matched by two directly connected nodes. This is typical of Web search. Considering that a database search user is more likely to retrieve the information that links two or more separated elements, a synthetic query set is also generated for IMDB. It has the same number of queries, 20, as the DBLP query set. Both synthetic query sets follow the same pattern. 50% of the queries are matched by two non-free nodes that are not directly connected. 20% of the queries cover three or more non-free nodes. The remaining queries can be matched by either a single node or two directly connected nodes.

The comparison in terms of mean reciprocal rank is shown in Fig. 8. For queries obtained from the AOL user log, the effectiveness of CI-RANK and SPARK, is close (0.85 and 0.79 respectively), both better than BANKS. For synthetic queries on both IMDB and DBLP, the mean reciprocal ranks of SPARK and BANKS are about 0.5, much lower than that of CI-Rank. The reason is that for most user log queries, the answers are directly connected non-free nodes, with no free nodes required. Only 11.4% of them require free nodes to connect the matched non-free nodes. For synthetic DBLP queries, this percentage is chosen to be 50%. This validates the effectiveness of our ranking function in selecting the free nodes to connect non-free nodes.

Fig. 9 shows the comparison in terms of precision. The precision of CI-Rank is higher than 0.9 in three different experiments. SPARK and BANKS also have high precision values, which are more than 0.85 for IMDB and 0.75 for DBLP. The difference in precision between CI-Rank and SPARK can be primarily attributed to those long queries that match three or more non-free nodes.

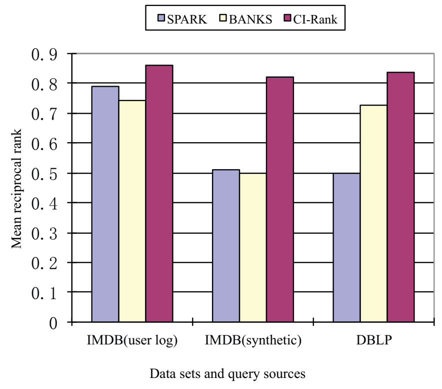


Fig. 8. Comparison of mean reciprocal rank

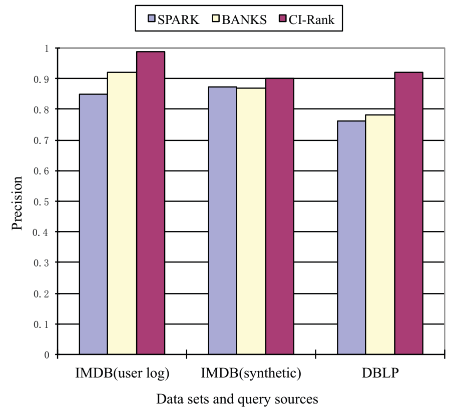


Fig. 9. Comparison of precision

C. Efficiency

We first show the efficiency comparison between the naive algorithm and the branch and bound algorithm. Because the naive algorithm can easily run out of memory when the whole datasets are used in their entirety, we obtain uniformly samples of the original datasets, with the size of each being 10% of the original. The experiments are performed on the sample datasets, and the results are presented in Fig. 10. It is evident that the branch and bound algorithm significantly outperforms the naive one.

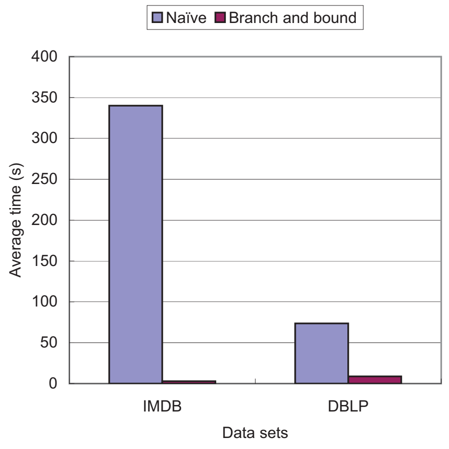


Fig. 10. The efficiency of the naive algorithm and the branch and bound algorithm

The evaluation of the index is performed on the full datasets. The search time for the answer tree is measured for different maximal tree diameters D, and the results are presented in Fig. 11 and 12. Each result is the average search time for the top-5 answers for the IMDB and DBLP queries. It is evident from these two graphs that the index reduces the search time considerably on both datasets. When the maximal tree diameter D decreases, the search time generally drops. For all maximal tree diameters, the average search time is less than 10 seconds when the index is used. Note that when the maximal tree diameter D is set to a small value (e.g., 2 or 3), the number of the full answer trees for some queries is less than 5, the value of k in top-k search.

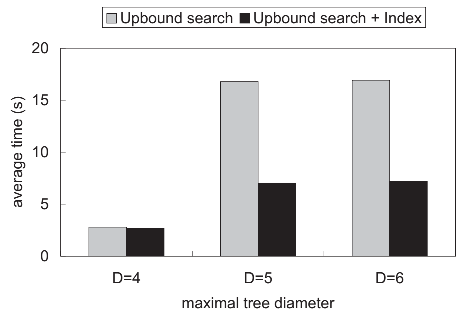


Fig. 11. The average search time for IMDB queries

VII. Related Work

Keyword search has been extensively studied in the literature ([1], [9], [7], [3], [12], [13], [15], [2], [18]). See [19] for an excellent survey. We have examined a few representative

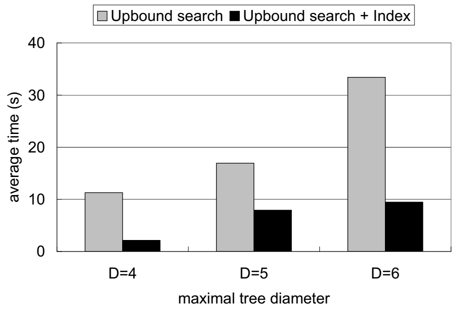


Fig. 12. The average search time for DBLP queries

approach in Section II. Here we briefly mention a couple other studies that are related to our work.

Instead of generating answer tree, EASE [11] looks for the most relevant subgraph, constrained by a maximum radius specified by the user. The ranking function in EASE is still IR-style. Thus, it does not consider the node importance.

CEPS [16] uses random walk with restarts (RWR), a variant of the random walk model, to find the best center-piece subgraph for a given graph and several pre-selected query nodes. The result of this algorithm is a subgraph rather than a tree, which is different from the query answer form used in our studies. A problem with this approach is that the query nodes are pre-selected in this model. For keyword search in databases, every input keyword will hit a lot of non-free tuples. It is unclear how to best choose candidates from all non-free tuples as the query nodes.

VIII. Conclusions and Future Work

We have proposed CI-Rank, a new approach for keyword search in databases. CI-Rank considers both the importance of individual nodes and the cohesiveness of the answer tree in a balanced way, and avoids a lot of the pitfalls of the ranking functions employed in previous methods. We presented a branch and bound algorithm to support the efficient generation of top-k query answers. The efficiency of the algorithm is further improved by star indexing. All solutions are implemented and evaluated against the IMDB and DBLP datasets. Real user query logs are employed in the evaluation.

In future work, we would like to study user preference adaptation. One possible direction is to consider how to improve the model such that user feedback can be used to adjust not only the importance values of the nodes, but also the weights of the edges of the database graph. Another direction is to study how to combine the importance-based ranking used in our approach and IR-style ranking to produce better ranking functions.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China Grants (No. 61070018, No. 60903108, No. 61003051), the Program for New Century Excellent Talents in University (NCET-10-0532), an NSERC Discovery Grant, the Independent Innovation Foundation of Shandong University (2009TB016), the SAICT Experts Program, and the Natural Science Foundation of Shandong Province of China under Grant No. 2009ZRB019RW. The authors would like to thank Ms. Jing Zhou, who helped with some of the experiments, and thank the anonymous reviewers, whose valuable comments helped improve this paper.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, page 5, Washington, DC, USA, 2002. IEEE Computer Society.

[2] A. Baid, I. Rae, J. Li, A. Doan, and J. F. Naughton. Toward scalable keyword search over relational data. *PVLDB*, 3(1):140–149, 2010.

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

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, and Chakrabarti. Keyword searching and browsing in databases using BANKS. In *ICDE*, page 431, Washington, DC, USA, 2002. IEEE Computer Society.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[6] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

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

[8] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1), 2008.

[9] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681. VLDB Endowment, 2002.

[10] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516. VLDB Endowment, 2005.

[11] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, New York, NY, USA, 2008. ACM.

[12] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, New York, NY, USA, 2006. ACM.

[13] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, New York, NY, USA, 2007. ACM.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[15] L. Qin, J. Yu, and L. Chang. Ten thousand sqls: Parallel keyword queries computing. *PVLDB*, 3(1):58–69, 2010.

[16] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, New York, NY, USA, 2006. ACM.

[17] K.-H. Vik, P. Halvorsen, and C. Griwodz. Evaluating Steiner-tree heuristics and diameter variations for application layer multicast. *Computer Networks*, 52(15):2872–2893, 2008.

[18] D. Xin, Y. He, and V. Ganti. Keyword++: A framework to improve keyword search over entity databases. *PVLDB*, 3(1):711–722, 2010.

[19] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Morgan and Claypool Publishers, 2010.