Noname manuscript No.
The VLDB Journal (2010) 18:31 - 60

Providing Built-in Keyword Search Capabilities in RDBMS

Guoliang Li · Jianhua Feng · Xiaofang Zhou · Jianyong Wang ·

Received on July 28, 2009

Abstract A common approach to performing keyword search over relational databases is to find the minimum Steiner trees in database graphs transformed from relational data. These methods, however, are rather expensive as the minimum Steiner tree problem is known to be NP-hard. Further, these methods are independent of the underlying relational database management system (RDBMS), thus cannot benefit from the capabilities of the RDBMS. As an alternative, in this paper we propose a new concept called *Compact Steiner Tree* (CSTree), which can be used to approximate the Steiner tree problem for answering *top-k* keyword queries efficiently. We propose a novel structure-aware index, together with an effective ranking mechanism for fast, progressive and accurate retrieval of *top-k* highest ranked CSTrees. The proposed techniques can be implemented using a standard relational RDBMS to benefit from its indexing and query processing capability. We have implemented our techniques in MYSQL, which can provide built-in keyword-search capabilities using SQL. The experimental results show a significant improvement in both search efficiency and result quality comparing to existing state-of-the-art approaches.

1 Introduction

Traditional query processing approaches on relational and XML databases are constrained by the query constructs imposed by query languages such as SQL and XQuery. First, the query languages themselves can be difficult to comprehend for non-database users. Second, these query languages require queries to be posed against the underlying, sometimes complex, database schemas. Third, although some applications can be provided on top of databases to facilitate easy access of the underlying data, it

Guoliang Li, Jianhua Feng, Jianyong Wang
Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China.
E-mail: {liguoliang,fengjh,jianyong}@tsinghua.edu.cn

Xiaofang Zhou
School of Information Technology and Electrical Engineering, The University of Queensland and NICTA Queensland Laboratory, Brisbane QLD 4072, Australia.
E-mail: zxf@itee.uq.edu.au

is rather difficult to devise a versatile application which can adapt to any underlying database. Fortunately, keyword search is proposed to provide an alternative means of querying relational databases, which is simple and familiar to most internet users. One important advantage of keyword search is that the users can search information without having to use a complex query language such as SQL or XQuery, or to understand the underlying data structures.

Keyword search is a proven and widely accepted query mechanism for textual documents and the World Wide Web, and the database research community has recently recognized the benefits of keyword search, starting to introduce keyword-search capabilities into relational databases [8, 28, 1], XML databases [24, 65], graph databases [33, 15, 25], and heterogenous data sources [39, 44]. The existing approaches of keyword search over relational databases can be broadly classified into two categories: those based on the candidate network [28, 26, 50] and others based on the minimum Steiner tree [8, 33, 15, 25]. The candidate-network-based methods use the schema graph to identify answers. They first generate candidate networks following the primary-foreign-key relationships, and then compute answers composed of relevant tuples based on candidate networks. The Steiner-tree-based methods use the data graph to compute relevant answers. They first model the tuples in a relational database as a data graph, where nodes are tuples and edges are the primary-foreign-key relationships. Steiner trees which contain all or some of input keywords are then identified to answer keyword queries. These methods on-the-fly compute answers by traversing the database graph to discover structural relationships. As the minimum Steiner tree problem is known to be NP-hard [20], a Steiner-tree-based approach can be inefficient, demanding new studies to find polynomial time solutions that can effectively approximate the minimum Steiner tree problem.

However, the existing RDBMS technologies have not been designed with supporting keyword search in mind; and the keyword search methods recently proposed by the database community are also largely independent of the underlying RDBMS. While the advantage for keyword-based search to be supported by the underlying RDBMS is quite clear, this integration task remains to be an open challenge. It is further complicated by other constraints such as user friendliness (using keyword search, not SQL queries) and no modification of any source code of an existing RDBMS.

To address this problem, we propose in this paper a polynomial time approximate solution for the minimum Steiner tree problem. A novel concept called *Compact Steiner Tree* (CSTree) is used to answer keyword queries more efficiently. We devise an effective structure-aware index and materialize the index in the form of relational tables. Our proposed techniques can be seamlessly integrated into any existing RDBMS such that the capabilities of the RDBMS can be explored to effectively and progressively identify the *top-k* relevant CSTrees using SQL statements. This key feature has been achieved without the need of changing any RDBMS source code. To the best of our knowledge, this is the first attempt to integrate structure-aware indices into RDBMS and use the capabilities of the RDBMS to support effective and progressive keyword-based search.

The main contributions of this paper include:

1. An efficient solution to approximate the minimum Steiner tree problem. A new concept called *Compact Steiner Tree* (CSTree), which can be used to approximate the Steiner tree problem for efficient answering of *top-k* keyword queries.

2. A structure-aware indexing mechanism for effective keyword search with embedded structural relationships and ranking score information. This indexing mechanism can be supported by the underlying RDBMS to utilize its capabilities for progressive and effective computation of *top-k* relevant CSTrees.
3. An implementation of providing built-in keyword search capabilities in MSQL and a comprehensive empirical performance study, which confirm the superior performance of the approach proposed in this paper over state-of-the-art methods.

The remainder of this paper is organized as follows. We formalize the minimum Steiner tree problem in Section 2. Section 3 proposes an approximate solution to approximate the Steiner tree problem. We introduce the concept of *Compact Steiner Tree* (CSTree) in Section 4, and discuss how to embed our proposed techniques into RDBMS so as to provide built-in keyword-search capabilities in Section 5. A ranking method by combining node weights and edge weights is proposed in Section 6. Section 7 reports our experimental results. We review major related work in Section 8 and conclude the paper in Section 9.

2 The Steiner-Tree-Based Approach

2.1 Database Graph

A relational database can be modeled as a database graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that there is a one-to-one mapping between a tuple in the database and a node in $\mathcal{V}$. $\mathcal{G}$ can be considered as a directed graph with two types of edges [8]: a forward edge $(u, v) \in \mathcal{E}$ iff there is a foreign key from u to v, and a back edge (v, u) iff (u, v) is a forward edge in $\mathcal{E}$. An edge (u, v) indicates a close relationship between tuples u and v (i.e., they can be directly joined together), and the introduction of two edge types allows differentiating the importance of u to v and vise versa. When such differentiation is not necessary for some applications, $\mathcal{G}$ becomes an undirected graph (i.e., the forward edge and its corresponding back edge are combined into one non-directional edge).

In order to support keyword search over relational data, $\mathcal{G}$ is typically modeled as a weighted graph, with a node weight $\omega(v)$ to represent the "prestige" level of each node $v \in \mathcal{V}$ and an edge weight $\omega(u, v)$ for each edge in $\mathcal{E}$ to represent the strength of the proximity relationship between the two tuples. In [8], $\omega(v)$ is set according to the indegree of v and can be transferred using PageRank style iterations in $\mathcal{G}$. We will discuss how to assign node weights in Section 6. While it is possible to set any desired value as the weight of an edge to reflect its importance (small values correspond to greater proximity between two nodes), they are typically determined according to edge types. For example, the weight of a forward edge (u, v) is defined based on the type of the primary-foreign-key in the schema, and default to 1; and for a back edge (v, u), $\omega(v, u) = \omega(u, v) * log_2(1 + \mathcal{D}_{in}(v))$, where $\mathcal{D}_{in}(v)$ denotes the indegree of node v [33].

For example, consider a publication database with four tables, "`paper`, `author`, `author-paper`, `paper-reference`." Figure 1 is a partial database graph for this database, containing 9 recent publications and 11 selected authors on the topic of keyword search in relational databases. Note that this graph is a simplified version for presentation purpose, omitting nodes for the tuples in relational table `author-paper` and relational table `paper-reference`, which only represent the primary-foreign-key relationships and do

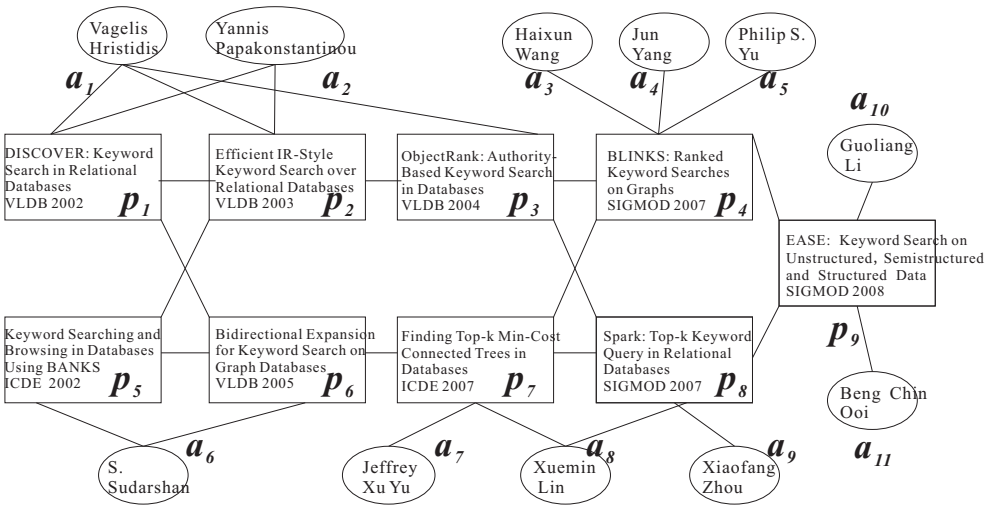

Fig. 1 A Running Example of Keyword Search over A Publication Database

not capture further useful information other than what the edges in the graph can already indicate. For presentation simplicity, we take the undirected graph as a running example in this paper and edge weights are not shown here. Note that, our method applies to directed graphs with any edge weight function.

2.2 The Steiner Tree Problem

To model the problem of identifying the *top-k* relevant answers from relational databases, we introduce the minimum Steiner tree problem [8] as follows.

Definition 1 MINIMUM STEINER TREE: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{V}' \subseteq \mathcal{V}$, $\mathcal{T}$ is a Steiner tree of $\mathcal{V}'$ in $\mathcal{G}$ if $\mathcal{T}$ is a connected subtree in $\mathcal{G}$ covering all nodes in $\mathcal{V}'$. Let $\varpi_\xi(\mathcal{T}) = \sum_{(u,v)\in\mathcal{T}} \omega(u,v)$. $\mathcal{T}$ is a minimum Steiner tree if $\varpi_\xi(\mathcal{T})$ is the minimum among all the Steiner trees of $\mathcal{V}'$ in $\mathcal{G}$.

This definition applies to both directed and undirected graphs (so trees are directed and undirected respectively). The minimum Steiner tree problem should not be confused with the minimum spanning tree problem: a Steiner tree of $\mathcal{V}'$ in $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ may contain nodes in $\mathcal{V} - \mathcal{V}'$. Next we introduce an extension of the minimum Steiner tree problem, allowing a node in $\mathcal{V}'$ be any representative of a group of nodes in $\mathcal{G}$.

Definition 2 MINIMUM GROUP STEINER TREE: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and groups $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n \subseteq \mathcal{V}$, $\mathcal{T}$ is a minimum group Steiner tree of these groups in $\mathcal{G}$ if $\mathcal{T}$ is a minimum Stiener tree that contains at least one node from each group $\mathcal{V}_i$, for $1 \leq i \leq n$.

A Steiner tree or a group Steiner tree, as originally defined, implies the minimum weight sum. As our interest in this paper is not limited to such a tree with the minimum weight but *top-k* such trees with the smallest values, we differentiate the concept of

Steiner tree (and group Steiner tree) in this paper from the minimum Steiner tree (and the minimum group Steiner tree).

The problem of finding the best answer of a keyword query in a database can be translated into the problem of finding the minimum group Steiner tree in the database graph [8]. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a database graph derived from an underlying database. Let $\mathcal{K}$ be a query containing keywords $k_1, k_2, \cdots, k_n$ against the database. Denote $\mathcal{V}_{k_i} \in \mathcal{V}$ for all nodes whose corresponding tuples contain keyword k_i, for $1 \leq i \leq n$ (note that such groups can be easily obtained by using an inverted index). The best answer to $\mathcal{K}$ is then represented by the minimum group Steiner tree $\mathcal{T}$ of $\mathcal{G}$, such that $\mathcal{V}(\mathcal{T}) \cap \mathcal{V}_{k_i} \neq \phi$ for $1 \leq i \leq n$.

The above approach returns only one solution which is represented by the minimum group Steiner tree. In the context of keyword search, however, a user is typically not satisfied with just one answer deemed by the system as the best; rather, they are interested in more answers ranked by their relevance to the query. We use the following example to explain the problem of finding *top-k* minimum group Steiner trees (i.e., finding *top-k* group Steiner trees with smallest $\varpi_\xi(\mathcal{T})$ values). We will discuss ranking-related issues in Section 6.

Example 1 Consider the graph in Figure 1. Given a node set $\mathcal{V}' = \{p_1, a_6\}$, the minimum Steiner tree is the subtree composed of nodes $\{p_1, p_6, a_6\}$. Given a keyword query $\{\texttt{Yu}, \texttt{graph}, \texttt{search}, \texttt{vldb}\}$, we have four sets of nodes that contain the two keywords, $\mathcal{V}_{\texttt{Yu}} = \{a_5, a_7\}$, $\mathcal{V}_{\texttt{graph}} = \{p_4, p_6\}$, $\mathcal{V}_{\texttt{search}} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_9\}$, and $\mathcal{V}_{\texttt{vldb}} = \{p_1, p_2, p_3, p_6\}$. The minimum Steiner tree is the subtree composed of nodes $\{a_5, p_3, p_4\}$. The *top*-2 minimum group Steiner trees are: the subtree composed of nodes $\{a_5, p_3, p_4\}$ and the subtree composed of nodes $\{a_7, p_7, p_6\}$. □

3 Using Minimum Path Weights

Finding *minimum (group) Steiner trees* is an important problem in many applications. These problems have been investigated extensively over the last three decades. These problems are known to be NP-hard [54]. In this section, we introduce the problem of *minimum path weight Steiner trees* (Section 3.1), and a polynomial time algorithm to solve this problem (Section 3.2).

3.1 Minimum Path Weight Steiner Tree

Different from traditional methods that identify the Steiner tree with the minimum edge weight, i.e., the sum of edge weights, we compute the Steiner tree with path weight. First, we introduce several notations. Given a tree $\mathcal{T}$ in graph $\mathcal{G}$, let $u \overset{\mathcal{T}}{\rightsquigarrow} v$ be a shortest path from node u to node v in $\mathcal{T}$, and $\omega(u \overset{\mathcal{T}}{\rightsquigarrow} v)$ be its path weight, which is the sum of the weight of each edge along the path.

Definition 3 Minimum Path Weight Steiner Tree: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{V}' \subseteq \mathcal{V}$. $\mathcal{T}$ is a minimum path weight Steiner tree of $\mathcal{V}'$ if (1) $\mathcal{T}$ is a Steiner tree of $\mathcal{V}'$; (2) Its path weight $\varpi_\rho(\mathcal{T}) = \min_{c \in \mathcal{T}} \{\sum_{v \in \mathcal{V}'} \omega(c \overset{\mathcal{T}}{\rightsquigarrow} v)\}$ is minimum among all Steiner trees of $\mathcal{V}'$, where c is a node in $\mathcal{T}$. We call c a *center node* of $\mathcal{T}$ w.r.t. $\mathcal{V}'$, if $c \in \mathsf{argmin}_{c' \in \mathcal{T}} \{\sum_{v \in \mathcal{V}'} \omega(c' \overset{\mathcal{T}}{\rightsquigarrow} v)\}$; and (3) There does not exist a subtree of $\mathcal{T}$ which is also a Steiner tree of $\mathcal{V}'$.

Note that a minimum path weight Steiner tree (MPWST) is not necessary a minimum Steiner tree in the traditional sense (i.e., using the sum of edge weights). For example, in Figure 1, suppose $\mathcal{V}'$={p_4, p_8, a_9, a_{10}, a_{11}}. Consider a Steiner tree with center node p_9 and containing nodes p_4, p_8, a_9, a_{10}, and a_{11}. Its path weight ϖ_ρ is 6 and its edge weight ϖ_ξ is 5. Also note that in a minimum path weight Steiner tree we do not keep those terminal nodes with degree one which are not in $\mathcal{V}'$. In other words, a minimum path weight Steiner tree must be "*minimum*", which has no subtree that is also a Steiner tree. For instance, consider another Steiner tree with nodes p_9, p_4, p_8, a_9, a_{10}, a_{11}, and a_5. Its path weight is also 6, but it is not a minimum path weight Steiner tree, as it has a subtree (by eliminating node a_5) which is also a Steiner tree.

Next we extend the above definition to define the concept of *minimum path weight group Steiner tree* (MPWGST) as follows.

Definition 4 Minimum Path Weight Group Steiner Tree: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and groups $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n \subseteq \mathcal{V}$. $\mathcal{T}$ is a minimum path weight group Steiner tree of these groups if (1) $\mathcal{T}$ is a group Steiner tree of these groups; (2) The path weight of $\mathcal{T}$ $\min_{c \in \mathcal{T}} \{\sum_{i=1}^{n} \omega(c \overset{\mathcal{T}}{\rightsquigarrow} v_i)\}$ is minimum among all group Steiner trees of $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n$, where c is a node in $\mathcal{T}$ and $v_i \in \mathcal{V}_i$ is a node in $\mathcal{T}$. We call c a *center node* of $\mathcal{T}$ w.r.t. $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n$, if $c \in \mathsf{argmin}_{c' \in \mathcal{T}} \{\sum_{i=1}^{n} \omega(c' \overset{\mathcal{T}}{\rightsquigarrow} v_i)\}$; and (3) There does not exist a subtree of $\mathcal{T}$ which is also a group Steiner tree of $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n$.

For example, in Figure 1, given three groups $\mathcal{V}_1$={a_1, a_2}, $\mathcal{V}_2$={a_5, a_6}, and $\mathcal{V}_3$={a_9, a_{10}}. Consider a group Steiner tree with center node p_4 and containing nodes p_3, a_1, a_5, p_9, and a_{10}. Its path weight is 5 and this group Steiner tree is a minimum path weight group Steiner tree. Note that a minimum path weight group Steiner tree must be "*minimum*", that is it has no subtree which is also a group Steiner tree. For instance, consider another group Steiner tree with nodes p_4, p_3, a_1, a_5, p_9, a_{10}, and a_4. Its path weight is also 5, but it is not a minimum path weight group Steiner tree, as it has a subtree (by eliminating node a_4) which is also a group Steiner tree.

3.2 Polynomial Algorithms

Now we consider how the MPWST problem can be solved using a polynomial time complexity algorithm.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a Steiner node set $\mathcal{V}' \subseteq \mathcal{V}$, a minimum path weight Steiner tree can be found in the following three steps: (1) the shortest paths from each node $s \in \mathcal{V}'$ to all other nodes in $\mathcal{G}$ are computed using, for example, the *Dijkstra* algorithm in $O(|\mathcal{V}'| * |\mathcal{V}|^2)$ time. (2) for each node $v \in \mathcal{V}$, a Steiner tree with center node v and containing the shortest paths from v to all the nodes in $\mathcal{V}'$ is constructed. (3) the minimum path weight Steiner tree can be found among the trees constructed in the previous step, with the complexity of $O(|\mathcal{V}'| * |\mathcal{V}|)$. Hence, the total complexity of the algorithm is $O(|\mathcal{V}'| * |\mathcal{V}|^2)$, which can be further improved, by using a more efficient shortest path algorithm [19], to $O\Big(|\mathcal{V}'| * \big(|\mathcal{V}| log(|\mathcal{V}|) + |\mathcal{E}|\big)\Big)$.

In a similar way, the MPWGST problem for a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set $\{\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_n\}$ of sets of nodes in $\mathcal{G}$ can also be solved in polynomial time. For each node $s \in \cup_{i=1}^{n} \mathcal{V}_i$, we first find the shortest paths from the single source node s to all other nodes in $\mathcal{G}$ by using the *Dijkstra* algorithm, with the complexity of $O(|\cup_{i=1}^{n} \mathcal{V}_i| * |\mathcal{V}|^2)$. Then,

for each node $v \in \mathcal{V}$, we construct a Steiner tree with center node v and containing the nodes from each $\mathcal{V}_i$ which has the minimum path weight to v among all the nodes in $\mathcal{V}_i$, with the complexity of $O\big(\sum_{i=1}^{n} |\mathcal{V}_i| * |\mathcal{V}|\big)$. Finally, we identify the minimum Steiner tree from the constructed Steiner trees. Thus, the total complexity is $O\big(|\cup_{i=1}^{n} \mathcal{V}_i| * |\mathcal{V}|^2 + \sum_{i=1}^{n} |\mathcal{V}_i| * |\mathcal{V}|\big)$, and the worse-case performance can be improved to $O\big(|\cup_{i=1}^{n} \mathcal{V}_i| * (|\mathcal{V}|log(|\mathcal{V}|) + |\mathcal{E}|) + \sum_{i=1}^{n} |\mathcal{V}_i| * |\mathcal{V}|\big)$ [19].

4 Compact Steiner Tree

While the algorithms of MPWST and MPWGST introduced in Section 3.2 are polynomial time algorithms, they are still time-consuming especially for large graphs. In addition, they are not designed for *top-k* keyword search either. In this section, we introduce a novel concept called *Compact Steiner Tree*, which can be used to facilitate fast retrieval of minimum path weight Steiner trees, and are highly effective in improving efficiency of *top-k* search.

4.1 Compact Steiner Tree

Clearly, not all nodes in a database graph are of equal importance to a term. For any term t, a node v in the graph can *contain* the term if t appears in the tuple of v; or *imply* the term if v does not contain t but there exists a path in the graph from v to any node that contains t; or *irrelevant* if it does not contain nor imply t. A node is *relevant* to t if it either contains or implies t. Intuitively, the relevance of a node u to a term t can be partially measured by the path weight of the shortest path from u to any node that contains t. Using the notion of the shortest path, we are ready to define *Voronoi partition* for the nodes relevant to a term.

Definition 5 VORONOI PARTITION: Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a database graph and t be a term. Let $\{v_1, v_2, \cdots, v_n\} \subseteq \mathcal{V}$ be the nodes containing t, and $U = \{u_1, u_2, \cdots, u_m\} \subseteq \mathcal{V}$ be the nodes implying t. A Voronoi partition of U for term t is defined as $U_1, U_2 \cdots U_n \subseteq U$ such that $\cup_{i=1}^{n} U_i = U$ and for any node $u \in U_i$, $\omega(u \overset{\mathcal{G}}{\rightsquigarrow} v_i) \leq \omega(u \overset{\mathcal{G}}{\rightsquigarrow} v_j)$, for $1 \leq j \leq n$.

When $u \in U_i$, u is said to be *dominated* by v_i with respect to t. We call v_i a *Voronoi node* of u with respect to t; and any shortest path from u to v_i a *Voronoi path*. As there could be multiple Voronoi nodes of u and t, we denote $\mathtt{vp}(u, t)$ as the set of Voronoi nodes of node u and term t. Note that such a Voronoi partition for a term can be effectively pre-computed and materialized [4]. We will introduce how to use Voronoi partitions to facilitate the keyword-based search in Section 5.

Example 2 Consider the graph in Figure 1. Given the keyword "`graph`" and the node set $\{p_4, p_6\}$ that contains the keyword, we have $\mathtt{vp}(a_6, \mathtt{graph}) = \{p_6\}$; $\mathtt{vp}(p_1, \mathtt{graph}) = \{p_6\}$; $\mathtt{vp}(a_5, \mathtt{graph}) = \{p_4\}$; $\mathtt{vp}(a_{11}, \mathtt{graph}) = \{p_4\}$. For Voronoi node p_4 of a_{11} and "`graph`", $a_{11} \overset{\mathcal{T}}{\rightsquigarrow} p_4 = a_{11} - p_9 - p_4$ is a Voronoi path for node p_4 and keyword "`graph`." Node p_6 is much more relevant to a_6 than p_4 for keyword "`graph`." Given node a_6 and keyword "`graph`," if we want to find the subtree with center node a_6 and containing

"graph," we expect to extend a_6 to p_6 with the edge of $a_6 - p_6$, instead of extending to p_4 with a long path of $a_6 - p_5 - p_2 - p_3 - p_4$. Thus, we can pre-compute and materialize *Voronoi paths*, e.g. $a_6 - p_6$ for "graph," which can facilitate retrieving Steiner trees as discussed in Section 5. □

Based on the notion of Voronoi partitions, we introduce the concept of *Compact Steiner Tree* as follows.

Definition 6 Compact Steiner Tree (CSTree): Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and groups $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n \subseteq \mathcal{V}$, $\mathcal{T}$ is a compact Steiner tree of these groups, if (1) $\mathcal{T}$ is a group Steiner tree of these groups; (2) There exists a node c in $\mathcal{T}$, such that for every group $\mathcal{V}_i (1 \leq i \leq n)$, c is in $\mathcal{V}_i$ or there exists a Voronoi path from node c to a node $v_i \in \mathcal{V}_i$ in tree $\mathcal{T}$. We call such node c a *center node* of CSTree $\mathcal{T}$ w.r.t. $\mathcal{V}_1, \mathcal{V}_2, \cdots \mathcal{V}_n$; and (3) $\mathcal{T}$ has no subtree satisfying (1) and (2).

A CSTree is said to be compact because all the Steiner nodes in such a tree must have the shortest path to a center node. If each $\mathcal{V}_i$ means all the nodes containing a unique term, then from the above definition a center node of a CSTree must be highly relevant to all n terms. For example, consider the graph in Figure 1. Given a keyword query {S., graph, BANKS}, the subtree with a center node a_6 and containing nodes $\{p_5, p_6\}$, is a Steiner tree as well as a CSTree, as a_6 is dominated by p_5 with respect to "BANKS" and also dominated by p_6 for "graph." The subtree with a center node p_5 and containing a_6 and p_2, p_3, p_4, is a Steiner tree but not a CSTree. This is because p_5 is not dominated by p_4 with respect to "graph." Obviously, the latter is less relevant than the former for this query.

Note that a CSTree is a special type of group Steiner tree. Therefore, the idea of computing minimum path weight group Steiner trees can also be applied here (called minimum path weight CSTrees). The intuition of finding *top-k* minimum path weight CSTrees, instead of finding *top-k* minimum path weight group Steiner trees, is that the CSTrees contains shortest paths for each node and keyword, and can provide more information than *top-k* minimum path weight group Steiner trees. For instance, to answer keyword query "S., BANKS, graph" over the graph in Figure 1, we prefer to identify CSTrees, $\mathcal{T}_1$, which has a center node a_6 and contains nodes $\{p_5, p_6\}$, and $\mathcal{T}_2$ which has a center node p_4 and contains nodes $\{a_5, p_3, p_2, p_5\}$ as the *top*-2 answers, instead of $\mathcal{T}_1$ and $\mathcal{T}_3$, where $\mathcal{T}_3$ is the subtree containing nodes $\{p_5, a_6, p_2, p_3, p_4\}$, since $\mathcal{T}_3$ is not a CSTree.

We compute CSTrees as the answer of keyword queries. Because firstly CSTrees have compact structures, with a center node dominated by the Steiner nodes which contain input keywords. Secondly, CSTrees can be effectively identified in decreasing order of their individual ranks, which will be further discussed in Section 5. Thirdly, Steiner trees can also be easily reconstructed from CSTrees, which will be demonstrated in the following subsections.

4.2 CSTrees vs. MPWGSTs

For keyword search, the user is likely to be interested in more than one answer ranked by their relevance to the query. Instead of ranking by the sum of edge weights of Steiner

trees, we propose to rank answers by path weights (i.e., using MPWGST). The purpose of introducing CSTree is to find a more efficient alternative to the MPWGST approach for *top-k* keyword queries. In this subsection, we study the relationship between MPWGST and CSTree. The minimum path weight CSTree and the minimum path weight MPWGST are simply called the minimum CSTree and the minimum MPWGST respectively in the remainder of this paper when there is no ambiguity.

First, let us explain why it can be prohibitively expensive to compute *top-k* MPWGSTs using the algorithm described in Section 3.2. Given groups $\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_n$, there exist many Steiner trees (even many more than $|\mathcal{V}| * \prod_{i=1}^{n} |\mathcal{V}_i|$). It is expensive to identify the *top-k* minimum MPWGSTs from this rather large number of possible Steiner trees. On the other side, for each CSTree, there exists a center node dominated by its Steiner nodes, and thus there are no more than $|\mathcal{V}|$ CSTrees. We can then construct the *top-k* CSTrees by adapting the algorithm of generating the minimum MPWGST in Section 3.2 as follows: for each node $s \in \cup_{i=1}^{n} \mathcal{V}_i$, we first find the shortest paths from the single source node s to all other nodes in $\mathcal{G}$ by using the *Dijkstra* algorithm. Then, for each node $v \in \mathcal{V}$, we select the node from each group $\mathcal{V}_i$ which has the minimum path weight to v among all the nodes in $\mathcal{V}_i$, and then construct a CSTree with center node v and containing the selected nodes. Finally, we identify the *top-k* CSTrees from the constructed CSTrees. The total complexity of this algorithm is $\mathrm{O}\Big(|\cup_{i=1}^{n} \mathcal{V}_i| * (|\mathcal{V}| log(|\mathcal{V}|) + |\mathcal{E}|) + \sum_{i=1}^{n} |\mathcal{V}_i| * |\mathcal{V}| + |\mathcal{V}| * log(k)\Big)$ [19]. Accordingly, we can effectively identify the *top-k* CSTrees. Moreover, we will propose the technique of progressively and effectively identifying the *top-k* CSTrees in Section 5.

One very important property of CSTrees is that the *top-k* MPWGSTs can be constructed from the *top-k* CSTrees. Suppose the *top-k* minimum CSTrees are $\{\mathsf{CST}_1, \mathsf{CST}_2, \cdots, \mathsf{CST}_k\}$. Initializing a set $\mathcal{S} = \{\mathsf{CST}_1, \mathsf{CST}_2, \cdots, \mathsf{CST}_k\}$. Consider CST_i with a center node c_i. For any keyword k_j and suppose node v is a Voronoi node of c_i and k_j, we can construct an MPWGST $\mathcal{ST}$ by replacing the path from c_i to v with another shortest path from c_i to any node containing k_j. (We can find the shortest path between two nodes using Dijkstra algorithm. If there are multiple shortest paths, we can find all of them by backtracking.) If $\varpi_\rho(\mathcal{ST}) < \max_{\mathsf{CST} \in \mathcal{S}}\{\varpi_\rho(\mathsf{CST})\}$, we add $\mathcal{ST}$ into $\mathcal{S}$ and remove the Steiner tree with the maximal cost from $\mathcal{S}$. If there is no such Steiner tree which satisfies $\varpi_\rho(\mathcal{ST}) < \max_{\mathsf{CST} \in \mathcal{S}}\{\varpi_\rho(\mathsf{CST})\}$, we have obtained the *top-k* minimum MPWGSTs in $\mathcal{S}$.

Example 3 Recall the graph in Figure 1. Given a keyword query $\{\texttt{Yu}, \texttt{Top-k}\}$, we first find the *top*-2 minimum CSTrees, which are the subtree composed of nodes $\{a_7, p_7\}$ and the subtree composed of nodes $\{a_5, p_4, a_7\}$. Then, we generate other MPWGSTs, such as the subtree composed of nodes $\{p_7, a_7, p_8\}$, the subtree composed of nodes $\{p_4, a_5, p_9, p_8\}$, and the subtree composed of nodes $\{p_4, a_5, p_3, p_8\}$. □

To summarize, given a node, there may be several MPWGSTs taking it as a center node, but there is one and only one CSTree taking it as a center node. The difference between MPWGSTs and CSTrees is: we group the MPWGSTs which share the same center node and take the CSTree from each group as the answer.

5 Indexing

To support efficient identification of *top-k* CSTrees from a very large database, it is highly desirable to use indexes to support the search and to utilize the power of the underlying RDBMS. In this section, we present an index that can support fast, effective and progressive computation of *top-k* relevant CSTrees.

5.1 Using a Relational Table to Store Voronoi Paths

Note that Voronoi partitions can be pre-computed, and thus Voronoi paths and their path weights can be pre-computed and indexed. We compute the score of a CSTree with respect to a query $\mathcal{K}$ by summing up the indexed path weights as shown below:

$$\psi(\text{CSTree}, \mathcal{K}) = \min_{c}\Big\{\sum\nolimits_{k_i \in \mathcal{K}} \omega\big(c \overset{\text{CSTree}}{\rightsquigarrow} p_i\big)\Big\} \tag{1}$$

where c is a center node in the CSTree and p_i is a Voronoi node for node c and keyword k_i.

To compute the path weight of a CSTree based on the above equation, we store Voronoi-paths and their path weights in a relational table:

$$\mathsf{Voronoi\text{-}path}(\mathsf{term}, \mathsf{node}, \mathsf{weight}, \mathsf{Voronoi\text{-}path}),$$

with $\langle \mathsf{term}, \mathsf{node} \rangle$ as the primary key. term is any keyword contained in the graph; node denotes the tuple that contains or implies the term; Voronoi-path is composed of all Voronoi paths from node to Voronoi nodes of node and term (as there could be multiple Voronoi paths with the same weight); and weight is the Voronoi-path weight.

5.2 Constructing the Relational Table

We can construct the Voronoi-path table off-line using a breadth-first traversal of the database graph as follows. For each node v in the graph, the terms which are contained in v are considered as highly relevant to v, as v itself is the corresponding Voronoi-node. Thus, we insert them into Voronoi-path table. Then, for each neighbor node of v, if the terms contained in the neighbor node are not contained by v, the neighbor node must dominate v for these terms and we insert such terms into Voronoi-path table; otherwise we discard such terms. Iteratively, we can construct the table by breadth-first traversing the graph. Note that when the path weight from a center node to a node u is larger than a given threshold (for example, we allow the maximal path weight from a center node to a leaf node be 4), we need not visit such nodes, since a path with large path weight will make the answer less meaningful.

We notice that the complexity of our algorithm is linear with the number of edges. Thus, the pre-computing cost is not proportional to the number of nodes. As a database graph is typically a sparse graph, the indexing cost is acceptable. We will experimentally prove this in Section 7.

5.3 Answering Keyword Queries Using the Relational Table

Based on the Voronoi-path table, given a keyword query $\{k_1, k_2, \cdots, k_n\}$, we can effectively and progressively identify the *top-k* highest ranked CSTrees from the Voronoi-path table as follows.

We first issue an SQL statement (MYSQL as an example) to find center nodes:

```
Select Node, Sum(Weight) As Cost
From Voronoi-Path (Table)
Where Term in (k_1, k_2, ..., k_n)
Group By Node
Having Count(Term) = n
Order By Cost
Limit k
```

We then construct the CSTrees by taking such identified nodes from the above SQL as center nodes and including all Voronoi-paths based on the Voronoi-path table. If users prefer MPWGSTs, we reconstruct MPWGSTs from CSTrees as described in Section 4.

Finally, as the results may have overlap, we need to remove the overlapped results through a postprocessing.

In addition, we can borrow the techniques for effective retrieving *top-k* answers from databases, such as Fagin Algorithm(FA), No Random Access algorithm(NRA), and Threshold Algorithm(TA) [16,31,3,30], to identify the *top-k* CSTrees on top of the Voronoi-path table[1]. This can offer significant improvement over existing methods to progressively answer *top-k* queries.

Example 4 Consider finding the *top*-2 results of query {Yu, Top-k} on the graph in Figure 1. We use the following SQL statement to find center nodes:

```
Select node, Sum(Weight) As Cost
From Voronoi-Path
Where Term in (Yu, Top-k)
Group By Node Having
Count(Term) = 2 Order By Cost
Limit 2.
```

Thus, we can use RDBMS capabilities to effectively identify the *top*-2 highest ranked CSTrees: the subtree with center node p_7 and containing a_7; and the subtree with center node p_4 and containing a_5 and p_7. □

5.4 Supporting the "OR" Semantics

The above method only considers the "AND" predicate and they can be rather inefficient for answering disjunctive queries. In this section, we introduce a method to

[1] Some RDBMSs have implemented such techniques. If an RDBMS does not provide such feature, we can implement TA algorithms using UDF to provide such feature.

Table 1 Voronoi-Path Table

Term	Node	Weight	Voronoi-path
Top-k	p_7	**0**	p_7
Top-k	p_8	0	p_8
Top-k	p_4	1	p_4-p_7
Yu	a_7	0	a_7
Yu	a_5	0	a_5
Yu	p_7	**1**	p_7-a_7
Yu	p_4	1	p_4-a_5
...	...	...	...

support the "OR" predicate, which does not need to modify the algorithms for identifying the *top-k* relevant answers in terms of disjunctive queries. Instead, we devise a technique of *transforming* the graph to support the "OR" predicate. We introduce τ_{k_i}, a real number that represents the penalty of missing keyword k_i. For example, given two connected subtrees $\mathcal{T}$ and $\mathcal{T}'$, $\mathcal{T}$ contains input keywords $\{k_1, k_2, \cdots, k_n\}$ and $\mathcal{T}'$ contains input keywords $\{k_1, k_2, \cdots, k_{i-1}, k_{i+1}, \cdots, k_n\}$. If $\varpi_\xi(\mathcal{T}) - \tau_{k_i} > \varpi_\xi(\mathcal{T}')$, we prefer $\mathcal{T}'$ to $\mathcal{T}$. This is because the cost of $\mathcal{T}$ minus the penalty of k_i is still larger than that of $\mathcal{T}'$. As different terms in the graph are different in importance, we should assign different penalty. We can set the penalty based on the inverse document frequency (idf) of a keyword. For example, we can set $\tau_{k_i} = \mathsf{ew}_{max} * \frac{ln(1+\mathsf{idf}_{k_i})}{ln(1+\mathsf{idf}_{max})}$, where ew_{max} denotes the maximal edge weight allowed in an answer, and idf_{k_i} denotes k_i's idf and idf_{max} denotes the maximal idf among all keywords in $\mathcal{G}$. Note that it is not easy to assign a good value of ew_{max}, and we need to tune the parameter to achieve high performance. In our experiment, our method achieves high performance when $\mathsf{idf}_{max} = 4$.

Accordingly, we can transform the graph to support the "OR" predicate as follows: For each node n in the graph, if n does not contain input keyword k_i, we add a node n_{k_i} and an edge (n, n_{k_i}) with the edge weight of τ_{k_i} into the graph. Consequently, we can apply any algorithm/method supporting the "AND" predicate on the *transformed* graph to support the "OR" predicate. Note that, actually, we need not add the nodes to change the graph structures. Instead, if node n implies or contains term t, we only keep the corresponding Voronoi paths in the Voronoi-path table, the weights of which are no larger than τ_t; if n is irrelevant to t, we set the weight as τ_t and Voronoi path as `NULL`. Thus, we need not modify the graph structures to support the "OR" predicate.

5.5 Reducing the Index Size

Recall the Voronoi-path table, it has $\mathcal{O}(m * n)$ records, where m is the number of nodes and n is the number of distinct keywords, and thus the index will be very large. However, we do not need to maintain all of these records. For each node, we only maintain the keywords highly relevant to the node. That is, we will not keep the keywords contained in nodes having large distance with a center node, since such nodes will make structure less compact and lead to the answer less meaningful. In the experiment, we only keep the keywords in the nodes that have the shortest path to a center node with path weight within four. Thus, the number of records is much smaller than $\mathcal{O}(m * n)$.

For the "AND" semantics, we can use the above SQL to answer keyword queries. For the "OR" semantics, we need to make a minor change and use the following SQL-like statement:

SELECT Node, (SUM(Weight-τ) + $\tau_{\mathcal{K}}$) AS Cost
FROM Voronoi-Path (Table)
WHERE Term in $(k_1, k_2, \cdots, k_n)$
GROUP BY Node
ORDER BY Cost
LIMIT k

where $\tau_{\mathcal{K}} = \sum_{k_i \in \mathcal{K}} \tau_{k_i}$ is the sum of penalty for missing all keywords, and τ is the penalty for missing the keyword in the record. In the index, for a record with keyword k_i, we store Weight-τ_{k_i} to replace Weight.

6 Ranking

So far the minimum Steiner tree problem and its various variations only consider the edge weight, but ignore the node weight. As not all nodes that contain or imply a term are equally important to the term, in this section we consider both node weights and edge weights to rank an answer.

6.1 Node Weight

There exist a rich body of literature dealing with node ranking in graphs, such as PageRank [9], HITS [35], and SALSA [38]. While these work mainly from the information retrieval domain can be adopted to assign node weights in a database graph, they only evaluate, however, the prestige (or authority, importance) of nodes in the graph. For example, the weight of node v, $\omega(v)$, is set to $\mathsf{PR}(v)$, which can be defined as:

$$\mathsf{PR}(v) = (1-d) + d * \left(\frac{\mathsf{PR}(u_1)}{\mathcal{D}_{out}(u_1)} + \cdots + \frac{\mathsf{PR}(u_m)}{\mathcal{D}_{out}(u_m)} \right) \tag{2}$$

where $\mathsf{PR}(v)$ is the prestige of node v; $\mathsf{PR}(u_i)$ is the prestige of node u_i which directly links to node v; $\mathcal{D}_{out}(u_i)$ is the outdegree of node u_i; and d is a damping factor that can be set between 0 and 1. In our experiment, d is set to 0.85. These traditional methods do not take into account term frequencies and inverse document frequencies in assigning node weights. To address this problem, ObjectRank [5] and EntityRank [10] proposed to rank an object or entity using the authority transfer paradigm and the hub framework. They can effectively rank a node in the graph. In this paper we seamlessly incorporate node weight and path weight to rank a tree-structure answer. We first discuss a scoring function to assign node weights by integrating PageRank and term importance scores defined by term frequency and inverse document frequency ($\mathsf{tf} \cdot \mathsf{idf}$) as below.

$$\omega(v|k) = \alpha * \omega(v) + (1-\alpha) * \mathcal{R}(v,k) \tag{3}$$

where

$$\mathcal{R}(v,k) = \begin{cases} ln\big(1 + \mathsf{tf}(v,k)\big) * ln\big(\mathsf{idf}(k)\big) & v \text{ contains } k \\ \max_{p \in \mathsf{vp}(v,k)} \{\mathcal{P}(v \overset{\mathcal{G}}{\rightsquigarrow} p) * \mathcal{R}\big(p,k\big)\} & v \text{ implies } k \end{cases} \tag{4}$$

Table 2 Structure-Aware Table

Term	Node	EdgeWeight	NodeWeight	Voronoi-path
Graph	p_4	0	$\omega(p_4)$+$ln2$*$ln10$	p_4
Graph	p_6	0	$\omega(p_6)$+$ln2$*$ln10$	p_6
···	···		···	···
Top-k	p_4	1	$\omega(p_4)$+$\frac{1}{5}$*$ln2$*$ln10$	p_4-p_7
···	···	···	···	···
Yu	p_7	1	$\omega(p_7)$+$ln2$*$ln10$	p_7-a_7
Yu	p_4	1	$\omega(p_4)$+$ln2$*$ln10$	p_4-a_5
···	···	···	···	···

$$\mathcal{P}\big(v \rightsquigarrow p\big) = \begin{cases} \prod\limits_{(x,y)\in v \overset{\mathcal{G}}{\rightsquigarrow} p} \frac{1}{\mathcal{D}_{in}(y)} & \text{directed} \\ \prod\limits_{(x,y)\in v \overset{\mathcal{G}}{\rightsquigarrow} p} \frac{1}{\mathcal{D}(y)} & \text{undirected} \end{cases} \tag{5}$$

$\omega(v|k)$ is the node weight of v w.r.t. term k by integrating node prestige $\omega(v)$ and $\mathsf{tf}\cdot\mathsf{idf}$ based relevance $\mathcal{R}(v,k)$, which evaluates the relevance between v and k. α is a tuning parameter to differentiate the importance of the node prestige and $\mathsf{tf}\cdot\mathsf{idf}$ based score, and is usually set to 0.8. $\mathsf{vp}(v,k)$ is the set of Voronoi nodes for node v and keyword k. Note that $p \in \mathsf{vp}(v,k)$ must contain k, thus $\mathcal{R}\big(p,k\big)$ can be computed based on tf and idf. $v \overset{\mathcal{G}}{\rightsquigarrow} p$ denotes the directed (undirected) path from v to p. $\mathcal{P}\big(v \overset{\mathcal{G}}{\rightsquigarrow} p\big)$ denotes the probability of randomly walking from v to p. $\mathsf{tf}(v,k)$ denotes the term frequency of k in v. $\mathsf{idf}(k)$ denotes the inverse document frequency of k in the database.

Note that traditional $\mathsf{tf}\cdot\mathsf{idf}$ based methods only consider the node which *contains* a given keyword. However, the node which *implies* a keyword is also relevant to the keyword. For example, consider p_7 in Figure 1, although p_7 does not contain "`keyword search`", it should be relevant to them, as it cites (is cited by) papers which contain the keywords. Alternatively, we also score the node which implies a keyword based on the notion of Voronoi partition and compute the extended $\mathsf{tf}\cdot\mathsf{idf}$ score as described in Equation 4. Note that $\mathcal{P}\big(n \overset{\mathcal{G}}{\rightsquigarrow} p\big)$ is a damping factor for node v implying keyword k. Thus, we incorporate the PageRank score and the extended $\mathsf{tf}\cdot\mathsf{idf}$ score into the scoring function as described in Equation 3, which can effectively measure the importance of a node and its relevance to a given keyword.

Note that node prestige is very important in database structures (capturing the primary-foreign-key relationship between tuples). Take the DBLP dataset as an example. Although some papers discussing "`Modeling Multidimensional Databases`" or "`Data Cube`" do not contain the keyword "`OLAP`" in their titles (not even in abstracts), they may still constitute relevant and potentially important papers in "`OLAP`" since they may be referenced by other papers in "`OLAP`" or written by the authors who authored other important "`OLAP`" papers. Based on the links between papers (citations), we can evaluate the prestige values, which can then be used for effective node ranking. The $\mathsf{tf}\cdot\mathsf{idf}$ score is also important for node weight. As the more keywords contained in the nodes, the more likely the nodes are relevant to the keyword queries. Note that an edge weight is also important to the ranking mechanism, as the answers with compact structure are most relevant to the query. We integrate the three aspects to effectively rank the query results. We will experimentally prove that node weight and edge weight can improve the result quality in Section 7.

Example 5 Consider keyword "graph" and nodes $\{p_4, p_6\}$ containing "graph" in Figure 1. $\mathcal{R}(p_6, \texttt{graph}) = ln2 * ln10$. $\mathsf{vp}(a_6, \texttt{graph}) = \{p_6\}$.

$\mathsf{vp}(p_2, \texttt{graph}) = \{p_6\}$.

$\mathcal{P}(a_6 \overset{\mathcal{G}}{\rightsquigarrow} p_6) = \frac{1}{4}$.

$\mathcal{P}(p_2 \overset{\mathcal{G}}{\rightsquigarrow} p_6) = \frac{1}{3} * \frac{1}{4} = \frac{1}{12}$.

$\mathcal{R}(a_6, \texttt{graph}) = \frac{1}{4} * \mathcal{R}(p_6, \texttt{graph}) = \frac{1}{4} * ln2 * ln10$.

$\mathcal{R}(p_2, \texttt{graph}) = \frac{1}{12} * \mathcal{R}(p_6, \texttt{graph}) = \frac{1}{12} * ln2 * ln10$.

Similarly, we can compute the node weights of other nodes, as shown in Table 2. □

6.2 Combining Node Weight and Path Weight

By integrating the node weight and edge weight, we propose a more effective ranking function Equation 6 to replace Equation 1 as follows:

$$\begin{aligned}\psi(\text{CSTree}, \mathcal{K}) &= \textsf{Node-Score} + \beta * \textsf{Edge-Score} \\ &= \max_c \Big\{ \sum\nolimits_{k_i \in \mathcal{K}} \omega(c|k_i) + \beta * \sum \omega(c \overset{\text{CSTree}}{\rightsquigarrow} p_i) \Big\} \\ &= \max_c \Big\{ \sum\nolimits_{k_i \in \mathcal{K}} \Big(\omega(c|k_i) + \beta * \omega(c \overset{\text{CSTree}}{\rightsquigarrow} p_i) \Big) \Big\}\end{aligned} \tag{6}$$

where c is a center node in CSTree, p_i is a Voronoi node for node c and keyword k_i, and β is a tuning parameter. Obviously, the larger Node-Score, the more relevant between CSTree and $\mathcal{K}$ from IR point of view; the smaller Edge-Score, i.e., the overall path weight of CSTree, the more meaningful of CSTree from the DB viewpoint, thus β is a negative real number and usually set to -0.8. Note that $\omega(c \overset{\text{CSTree}}{\rightsquigarrow} p) = \omega(c \overset{\mathcal{G}}{\rightsquigarrow} p)$, which can be pre-computed as formalized in Equation 1. $\omega(r|k_i)$ can also be pre-computed and materialized as described in Equation 3.

To maintain $\omega(c \overset{\mathcal{G}}{\rightsquigarrow} p_i)$ and $\omega(r|k_i)$, we modify the Voronoi-path table to construct a structure-aware table, which is obtained with the addition of two attributes NodeWeight and EdgeWeight. The two attributes are used to index the node weight and edge weight respectively. Accordingly, we incorporate the ranking scores and the structural relationships (Voronoi-paths) into our indices as shown in Table 2, and thus our proposed index is structure-aware and can facilitate the *top-k* based keyword search. To effectively answer a keyword query, we issue an SQL statement (MYSQL as an example):

```
SELECT Node,SUM(NodeWeight+β*EdgeWeight) AS Cost
FROM Structure-Aware (Table)
WHERE Term in (k_1, k_2, ··· , k_n)
GROUP BY Node
HAVING COUNT(Term) = n
ORDER BY Cost DESC
LIMIT k
```

Obviously, the SQL statement can exactly identify the *top-k* highest ranked CSTrees. Accordingly, we can use the SQL capabilities of the RDBMS to effectively and progressively identify the *top-k* answers, instead of finding the answers by traversing the

graphs to discover structural relationships. Most importantly, we can easily incorporate our method into the existing RDBMS, and we need not modify any source code of RDBMS to support keyword-based search.

6.3 Supporting Dynamic Update

In terms of update issues, note that we do not need to reindex the whole graph from scratch. Instead we can incrementally update the index as follow.

(1) Node relabeling: That is the keywords contained in a node is updated. In this case, only the records that contain the relabeled node in the structure-aware table are updated. Suppose node n is relabeled. For each keyword w in the new label or the old label of n, if node n is included in the `Node` attribute of a record,
   - If both the new label and the old label contain w, we do not need to update.
   - If the new label contains w and the old label does not contain w: (i) if there is no such a record with primary key $\langle w, n\rangle$, we insert record $r = \langle w, n, 0, w_n, vp\rangle$ into the index, where w_n is the node weight and vp is a Voronoi path; (ii) otherwise, we update such record with the above Voronoi path and weights.
   - If the new label does not contain w and the old label contains w: (i) if there is a record with primary key $\langle w, n\rangle$, we update the Voronoi path in the record to the path from n to the nearest neighbor of n that contains w, and change the node weight and edge weight; (ii) otherwise we delete such record.

   If node n is included in the `Voronoi path` attribute of a record,
   - If both the new label and the old label contain w, we do not need to update.
   - If the new label contains w and the old label does not contain w: (i) if there is a record with primary key $\langle w, n\rangle$ and a Voronoi path in the record is longer than the new Voronoi path vp for w and n, we update the Voronoi path to vp, and change the node weight and path weight; otherwise, we do not need to update; (ii) if there is no such a record, we insert record $r = \langle w, n, l, w_n, vp\rangle$ into the index, where vp is a Voronoi path for w and n, l is the path weight, and w_n is the node weight.
   - If the new label does not contain w and the old label contains w: (i) if there is a record with primary key $\langle w, n\rangle$ and a Voronoi node is n, we update the Voronoi path to a new Voronoi path vp for w and n, and change the node weight and path weight; (ii) otherwise, we do not need to update.

(2) Node insertion: That is we add a node n. Firstly, for each keyword w contained in n's neighbors (including n), we insert record $r = \langle w, n, l, w_n, vp\rangle$ into the index, where vp is a Voronoi path for w and n, l is the path weight, and w_n is the node weight. Secondly, for each keyword w contained in n, for each n's neighbor v, if there is a record with primary key $\langle w, v\rangle$, we update the corresponding Voronoi path to the new Voronoi path for w and v; otherwise, we insert record $r = \langle w, v, l, w_v, vp\rangle$ into the index, where vp is a Voronoi path for w and v, l is the path weight, and w_v is the node weight.

(3) Node deletion: That is we delete a node n. Firstly, we remove the records that contain node n in the `Node` attribute. Secondly, for the records that contain node n in the Voronoi path attribute, if the corresponding keywords are still implied by node n, we update the Voronoi path and the corresponding path weight and node weight; otherwise we remove such records.

(4) Edge insertion: That is we add an edge. Suppose the two nodes of the edge are u and v. For the records that contain the two nodes in the Voronoi path attribute, we update the Voronoi path and the corresponding path weight and node weight. For the records that contain the node $u(v)$ in the `Node` attribute, for each keyword w contained in the $u(v)$'s neighbor, if there is a record with primary key $\langle w, u(v)\rangle$, we update the Voronoi path and the corresponding path weight and node weight; otherwise, we insert $r = \langle w, u(v), l, w_{u(v)}, vp\rangle$ into the index, where vp is a Voronoi path for w and $u(v)$, l is the path weight, and $w_{u(v)}$ is the node weight.

(5) Edge deletion: That is we remove an edge. Suppose the two nodes of the edge are u and v. For the records that contain the two nodes in the Voronoi path attribute, we update the Voronoi path and the corresponding path weight and node weight. For each record r that contains the node $u(v)$ in the `Node` attribute, if the corresponding keyword w is contained in the $u(v)$'s neighbor, we update the Voronoi path and the corresponding path weight and node weight; otherwise, we remove the record.

We can implement a user-defined function (udf) to support dynamic update.

7 Experimental Study

We have implemented our method in MySQL 5.0.22[2], and used SQL statements in Section 6 to generate answers.

We compared search efficiency and result quality with existing state-of-the-art algorithms BLINKS [25] and EASE [44][3]. The datasets used include a publication database DBLP (http://dblp.uni-trier.de/xml/) and a movie database IMDB (http://www.imdb.com/). The DBLP dataset contains 475 MB raw data, which are translated into four tables as illustrated in Table 3. IMDB contains approximately one million anonymous ratings of 3900 movies made by 6040 users. Although the IMDB dataset has only 10, 000 nodes (users+movies), there are one million edges. Thus the dataset is a big graph, which is much denser than the DBLP dataset.

We model the databases as undirected graph and the edge weight is set to 1. The maximal path weight of a Steiner tree is set to 4. The elapsed time of indexing DBLP and IMDB is 2635 seconds and 278 seconds respectively. DBLP contains 1.4m distinct keywords and IMDB contains 466k distinct keywords. The sizes of the indices of DBLP and IMDB are respectively 1242 MB and 96 MB, compared with data sizes of 475 MB and 30 MB. Note that our method is only effective for sparse graph, and the index may be huge for dense graph.

All the algorithms were coded in JAVA. All the experiments were conducted on a computer with an Intel(R) Core(TM) 2@2.33GHz CPU, 2 GB RAM running Windows XP.

7.1 Search Efficiency

We evaluate search efficiency of various algorithms in this section. We used the average elapsed time to compare the performance of different algorithms. We selected one hundred keyword queries on each dataset with different numbers of query keywords. Table 4 gives several sample queries. Figure 2 illustrates the experimental results.

[2] In all the experiments, d is set to 0.85, α is set to 0.8, and β is set to -0.8.

[3] We implemented the two algorithms by ourselves.

Table 3 Data Sets

(*a*) DBLP Dataset

Tables	Attributes	# of Records
Author	AID, AName	630,751
Paper	PID, PName, Year, Booktitle	1,062,361
Author-Paper	AID, PID	2,616,736
Paper-Reference	PID, citedPID	112,304

(*b*) IMDB Dataset

Tables	Attributes	# of Records
User	UID, UserName, Gender, OID	6,040
Movie	MID, Title, Genres	3,883
Occupation	OID, Occupation	30
Rating	UID, MID, Score	1,000,209

Table 4 Several sample queries employed in the experiments

(*a*) Five sample queries on DBLP dataset

Queries
Information Retrieval Database
IR Database
DB IR XML
XML Relational Keyword Search
Data Mining Algorithm 2006

(*b*) Five sample queries on IMDB dataset

Queries
Lethal Weapon 4 academic starship
Police Academy 3 customer
Halloween 5 college academic
Love 45 tradesman love weapon
Robocop 3 college customer

We observe that our algorithms achieve much better search performance than the existing methods EASE and BLINKS. CSTree clearly yields high search efficiency as it does not need to identify answers by discovering the relationships between tuples in different relational tables on the fly, relying instead on SQL capabilities of the RDBMS to identify the answer. CSTree is two to four times faster than EASE as EASE involves a post-processing to extract Steiner graphs by eliminating non-Steiner nodes. Moreover, with the increase of the numbers of input keywords, the elapsed time of BLINKS increases dramatically, but the elapsed time of CSTree varies slightly and always achieves higher search efficiency. This remarkable achievement can be attributed to the structure-aware index used and the use of the power of RDBMS to identify answers in our approach, instead of discovering the Steiner trees by traversing the graph in other approaches. Although BLINKS uses indices to improve search efficiency, it involves traversing indices to discover the root-to-leaf paths for identifying the answers. For instance, with the IMDB dataset, CSTree takes less than 130 ms to answer keyword queries with six keywords; EASE takes 420 ms; BLINKS involves more than 1200 ms. Clearly, CSTree outperforms BLINKS by an order of magnitude and is also significantly faster than EASE. This comparison convincingly shows superior efficiency of our proposed techniques.

To better understand the performance of our algorithm, we searched for the *top-k* relevant answers and compared their corresponding costs (average elapsed time). Fig-

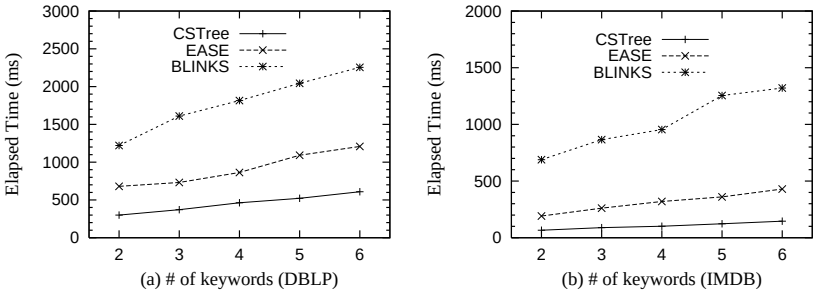

Fig. 2 Search Efficiency

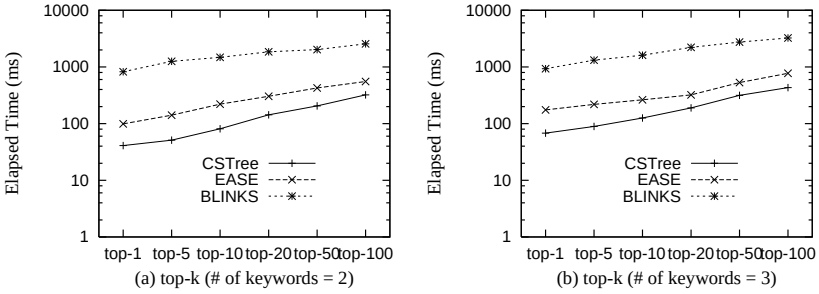

Fig. 3 Top-k Efficiency on DBLP Dataset

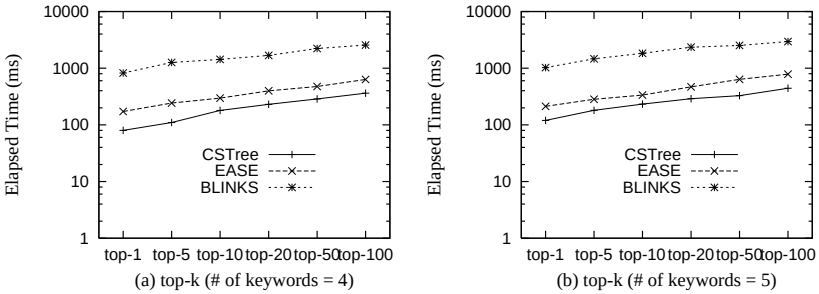

Fig. 4 Top-k Efficiency on IMDB Dataset

ure 3 and Figure 4 summarize the experimental results on the two datasets respectively. One can see that our method outperforms the existing methods EASE and BLINKS significantly. This is attributed to the threshold-based techniques [16] we adopted in this paper to identify the *top-k* answers, allowing us to progressively find the *top-k* relevant results. For example, in Figure 3 (a), CSTree takes 80 ms to identify the *top*-5 results, while EASE and BLINKS take more than 200 ms and 1000 ms respectively on the DBLP dataset.

7.2 Result Quality

This section evaluates the quality of search techniques in terms of accuracy and completeness using standard precision and recall metrics. There is no standard benchmark

for keyword search over relational databases. We enumerated all possible SQL queries for a set of keywords (as discussed in [23]) to generate our baseline query results, which are assumed as accurate and complete and used to compute precision and recall. For each query, we generated all possible SQL queries as follows. Firstly, we join the tables to formulate SQL queries and the where clauses of the SQL queries contain input keywords. Secondly, the tables that contain the keywords can be connected within a specific distance (We set the distance as 4). Thirdly, the tables that contain no keywords must be used to connect tables; otherwise we remove them. For the top-k queries, we evaluated the precision and recall by user study, and the answer relevance of a query is judged from discussions of the twenty people attending the user study. Precision measures search accuracy, indicating the fraction of results in the approximate answer that are correct, while recall measures completeness, indicating the fraction of all correct results actually captured in the approximate answer.

7.2.1 Search Accuracy

This section evaluates search accuracy (i.e., precision). Figure 5 gives the experimental results. This figure clearly demonstrates that our method achieves constantly higher search accuracy and outperforms the existing methods. CSTree leads BLINKS by about 20%-40% on various queries with different numbers of input keywords. This is because our ranking method incorporates both the PageRank and the $\mathsf{tf} \cdot \mathsf{idf}$ based IR technique and the structural information in DB perspective into the ranking function. CSTree also outperforms EASE about 5%-10% in terms of precision, as we take into consideration the node prestige to rank the answers while EASE ignores such feature. We note that the node prestige is very important to rank the query results. For example, consider the DBLP dataset. Given a keyword query, although some papers do not contain the input keywords, they are also very relevant to the query, as they may cite some papers or are cited by some papers which contain the input keywords. The same is true for movies in IMDB dataset.

As users are usually interested in the *top-k* answers, we used the metric of *top-k* precision to measure the ratio of the number of relevant answers among the first k answers with the highest scores of an algorithm to k. Figure 6 and Figure 7 illustrate the experimental results, which show that CSTree achieves higher precision than BLINKS and EASE on different values of k. Note that with the increase of k values, *top-k* precision of BLINKS drops dramatically while CSTree can always achieve high precision. This is because we employ a very effective ranking mechanism by considering PageRank, $\mathsf{tf} \cdot \mathsf{idf}$, and the structure information. CSTree outperforms EASE and BLINKS by approximately 15% and 30% respectively. This confirms the benefit of our ranking mancinism which combines the feature of PageRank, $\mathsf{tf} \cdot \mathsf{idf}$ based techniques, and structural relationships embedded in the database graph. Moreover, CSTree prunes away many less *meaningful* and less *compact* Steiner trees.

7.2.2 Search Completeness

This section evaluates the search completeness (i.e., recall). Figure 8 illustrates precision/recall curves obtained from our experiments. It shows that with the increase of recall, the precision of BLINKS drops down sharply while that of CSTree varies only slightly. This is because some Steiner trees with less compact structures are not

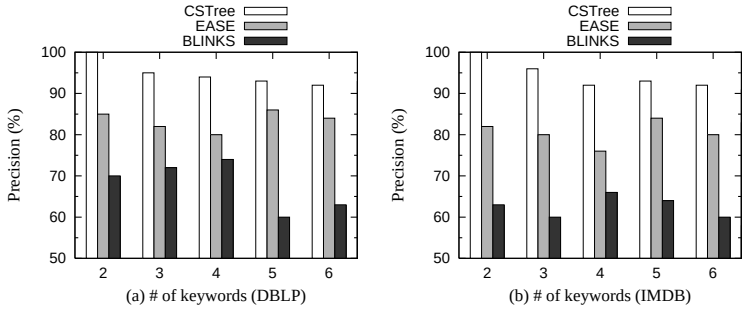

Fig. 5 Search Accuracy

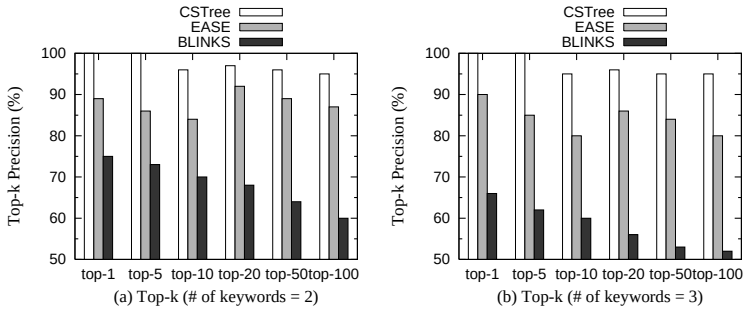

Fig. 6 Top-k Precision on DBLP Dataset

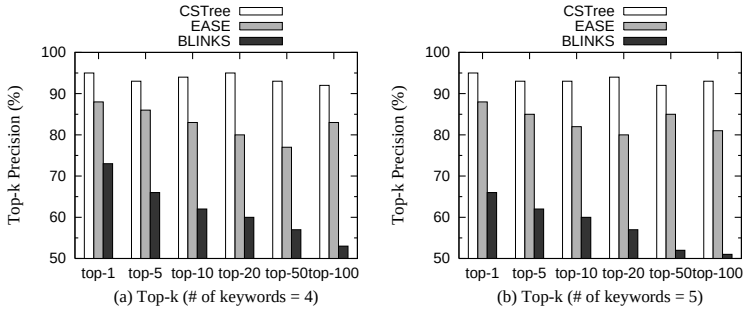

Fig. 7 Top-k Precision on IMDB Dataset

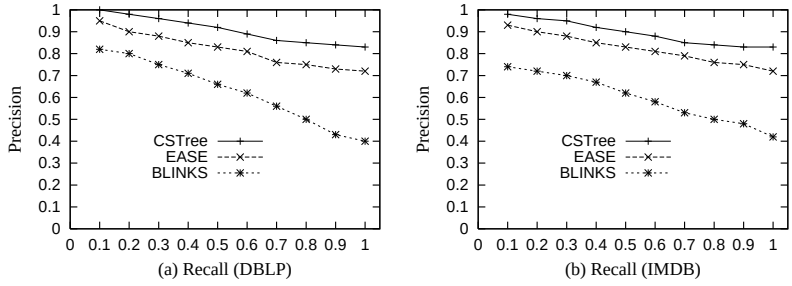

Fig. 8 Search Completeness

Table 5 Selected Keyword Queries for User Study

(*a*) Queries on DBLP dataset

	Queries
Q_1	Data Cube OLAP
Q_2	Similarly String Search
Q_3	XML IR Keyword Search
Q_4	Uncertain Probabilistic Data Approximate
Q_5	Data Mining Graph Jiawei Han

(*b*) Queries on IMDB dataset

	Queries
Q_6	Lethal Weapon 2 college comedy
Q_7	Police Academy 5 tradesman weapon
Q_8	Halloween 3 college
Q_9	Love 36 customer
Q_{10}	Robocop 1 academic

relevant to the queries, and CSTree prunes those incompact Steiner trees and identifies the most meaningful and relevant ones as the answers. Moreover, we employ an effective ranking technique to rank the query results.

7.3 Evaluating Result Quality by Human Judgement

To further evaluate result quality of different methods, we evaluated the query results by human judgement. We selected ten keyword queries on the two datasets as illustrated in Table 5. Answer relevance of the selected queries is judged from discussions of twenty researchers in our database group. As users are usually interested in the *top-k* answers, we employ *top-k* precision, i.e., the ratio of the number of answers deemed to be relevant in the first k results to k, to compare those algorithms.

We first computed the *top*-100 results for every query and compared the corresponding *top*-100 precision. Figure 9 illustrates the experimental results obtained. We observe that CSTree achieves much higher precision than EASE and BLINKS on the corresponding datasets. This is because we employ effective ranking techniques, which consider textual relevancy, structural compactness, and node prestige. For example, for query Q_1, although the paper entitled "`Modeling Multidimensional Databases`" does not contain the input keywords, it is a relevant paper to "`Data Cube`" and "`OLAP`." CSTree can find such answers but EASE and BLINKS cannot.

We then varied different values of k and evaluated the average *top-k* precision of the selected queries. The results of the average *top-k* precision for Q_1-Q_{10} are illustrated in Figure 10. As expected, CSTree consistently achieves high precision in all the queries, which is approximately 5-15% higher than EASE and 10-30% higher than BLINKS. This confirms the superiority of our proposed ranking techniques.

7.4 Evaluation on The TPC-H Dataset by Considering Keyword Distribution

We evaluated the three methods on the TPC-H dataset[4], which contains eight relational tables. We used the TPC-H schema, and generated the data by controlling the

[4] http://www.tpc.org/tpch/

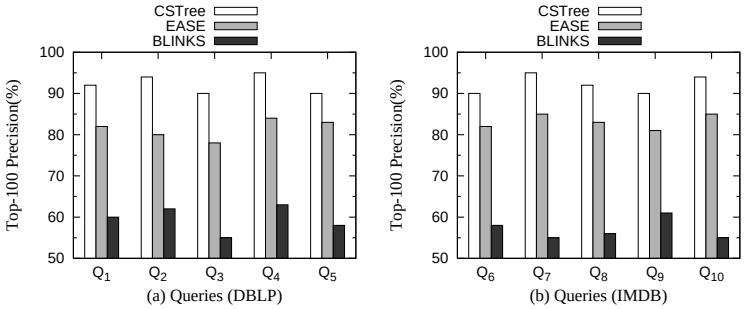

Fig. 9 Top-100 Precision by Human Judgement

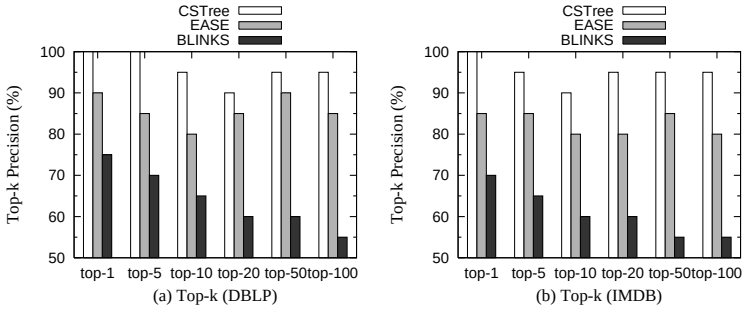

Fig. 10 Top-k Precision by Human Judgement

distribution of the number of occurrences of every keyword in a query [28]. For example, a keyword is contained in table LINEITEM with probability $\frac{1}{50}$, in table REGION with probability $\frac{1}{200}$, and in table PART with probability $\frac{1}{100}$. The dataset size is 400 MB and the index size is 1.4 GB.

We first evaluated top-k precision based on human judgement as described in Section 7.3 and Figure 11 shows the results. We can see that our method achieves high search quality. This is because CSTrees employed a better ranking function, which considers PageRank, $\mathsf{tf} \cdot \mathsf{idf}$, and the structural compactness. For example, CSTrees outperforms EASE and BLINKS by about 10%.

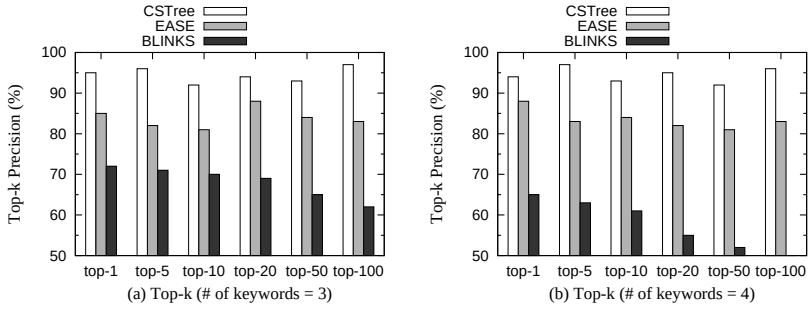

Fig. 11 Top-k Precision by Considering Keyword Distribution

We then evaluated the search performance. Figure 12 shows the results. We can see that even if on the dataset by considering keyword distribution, our method still achieves high performance. This is attributed to our indexing and search method, which uses database capabilities to efficiently find answers. For example, for queries with 3 keywords, CSTrees took less than 100 ms to find the answers, EASE took 200 ms, and BLINKS took more than 1000 ms.

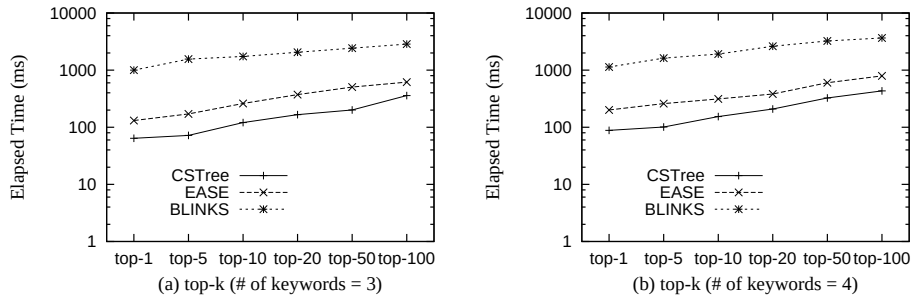

Fig. 12 Top-k Efficiency by Considering Keyword Distribution

7.5 Evaluation of Update

In this section, we evaluate the performance of dynamic updates on DBLP dataset and show the benefit of our proposed incremental method to update indexes. Initially, we selected 100 MB data as the original dataset, and for each time, we updated it by inserting 10 MB new data. We compared the running time between the incremental method and the method that rebuilds the index from scratch. Figure 13 shows the experimental results. We can see that the incremental method outperforms the method that rebuilds the index from scratch. Because the incremental method does not need to rebuild the index from scratch.

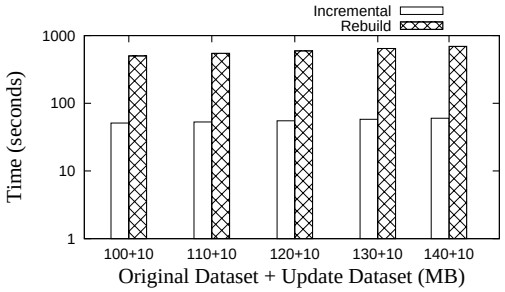

Fig. 13 Evaluation of update on DBLP dataset

8 Related Work

Existing approaches to support keyword search over relational databases can be broadly classified into two categories: those based on the candidate network [28, 26] and others based on the Steiner tree [8, 33, 15]. The former uses schema graph to compute the answers and the latter uses the data graph to identify the answers.

DISCOVER-I [28], DISCOVER-II [26], BANKS-I [8], BANKS-II [33] and DBXplorer [1] are examples of keyword search systems built on top of relational databases. DISCOVER and DBXplorer output trees of tuples connected through the primary-foreign-key relationship that contain all input keywords. DISCOVER-II considers the problem of keyword proximity search in terms of the disjunctive semantics, unlike DISCOVER-I which only considers the conjunctive semantics. BANKS-I identifies connected trees, namely Steiner trees, in a labeled graph by using an approximation of the Steiner tree problem. BANKS-II is an improvement on BANKS-I, introducing a novel technique of bidirectional expansion to improve the search efficiency.

Liu et al. [47] proposed a new ranking strategy to solve the search effectiveness problem for relational databases. The strategy employs phrase-based and concept-based models to improve search quality. Object-Rank [5] improves search quality by adopting a hub-and-authority style [35] ranking method when answering keyword search over relational databases. While this method is effective in ranking objects, pages and entities, it cannot effectively rank tree-structured results (e.g., Steiner trees), as it does not consider structure compactness of an answer in its ranking function. Kimelfeld et al. [34] demonstrated keyword proximity search in relational databases, which shows that the answer of keyword proximity search can be enumerated in ranked order with polynomial delay under data complexity. Golenberg et al. [22] presented an incremental algorithm for enumerating subtrees in an approximate order which runs with polynomial delay and can find all the *top-k* answers. Sayyadian et al. [55] introduced schema-mapping into keyword search, proposing a method to answer keyword search across heterogenous databases. Ding et al. [15] employed a technique of dynamic programming to improve search efficiency via identifying Steiner trees. Guo et al. [23] used data topology search to retrieve meaningful structures from much richer structural data (e.g., biological databases). Markowetz et al. [51] studied the problem of keyword search over relational data streams. They proposed to generate operator trees and several optimization techniques using mesh to answer keyword queries over streams. Luo et al. [50] proposed a new ranking method that adapts state-of-the-art IR ranking functions and principles into ranking tree-structured results composed of joined database tuples. He et al. [25] proposed a partition-based method to improve the search efficiency by means of the BLINKS index. Dalvi et al. [14] proposed a new concept of "supernode graph" to address the problem of keyword search on graphs that may be significantly larger than memory.

Koutrika et al. [37] proposed to use clouds over structured data (data clouds) to summarize the results of keyword searches over structured data and to guide users to refine searches. Zhang et al. [68] and Felipe et al. [17] studied keyword search on spatial databases by combining inverted lists and R-tree indexes. Tran et al. [62] studied keyword search on RDF data. Tao et al. [60] proposed to find co-occurring terms of input keywords, in addition to the answers, in order to provide users relevant information to refine the answers. Qin et al. [52] studied three different semantics of m-keyword queries, namely, connect-tree semantics, distinct core semantics, and distinct root semantics, to answer keyword queries in relation databases. The efficiency is achieved by

new tuple reduction approaches that prune unnecessary tuples in relations effectively followed by processing the final results over the reduced relations. Chu et al. [12] attempted to combine forms and keyword search and used keyword search techniques to design forms. Yu et al. [67] and Vu et al. [63] studied keyword search over multiple databases in P2P environment. They emphasize on how to select relevant database sources. Chen et al. [11] gave an excellent tutorial of keyword search in XML data and relational databases. The recent integration of DB and IR was reported in [2, 7, 64, 58]. We studied type-ahead search in relational databases [42], which allows searching on the underlying relational databases on-the-fly as users type in query keywords. Ji et al. [32] studied fuzzy type-ahead search on a set of tuples/documetns, which can on-the-fly find relevant answers by allowing minor errors between input keywords and the underlying data. Richardson and Domingos [53] proposed to combine page content and link structure. They used a probabilistic model of the relevance of a page to a query to combine the two factors. Different from their method, we use random walk to compute node weights by integrating PageRank and term importance scores defined by term frequency and inverse document frequency ($\mathsf{tf} \cdot \mathsf{idf}$).

Regardless their apparent differences, the existing methods are based on computing Steiner trees composed of relevant tuples by discovering the structural relationship of primary-foreign-keys on the fly. As this is an NP-hard problem, there is a need to find efficient and effective approximate solutions for the Steiner tree problem in large graphs. A number of approximation algorithms with polynomial time complexity have been proposed (e.g., [21] and [54]). Robins et al. [54] proposed a good polynomial time algorithm to find minimum weight Steiner tree with an approximation ratio around 1.55. Different from these studies, in this paper we focus on using SQL queries to find compact Steiner trees.

In this paper, we propose an approximate algorithm which has polynomial running time and returns solutions that are not far from an optimal solution. Moreover, departing from the traditional approaches based on Steiner trees, we introduce a novel concept of *Compact Steiner Tree*, or CSTree, to answer keyword queries. CSTrees can be effectively and progressively generated in decreasing order of their individual ranks. This proposal is orthogonal to BLINKS [25]. Firstly, BLINKS maintains two indices of Keyword-NodeList and Node-KeywordMap. CSTree only maintains a structure-aware index, which is similar to Keyword-NodeList. Our structure-aware index keeps the root-to-leaf path and its path weight. Keyword-NodeList in BLINKS only keeps the nodes instead of the paths. Thus, BLINKS must traverse the Keyword-NodeList index to identify the root-to-leaf paths, which can lead to low efficiency. Secondly, we emphasize on using the database capabilities for keyword search and push our techniques into database systems, and evidently, BLINKS cannot. Thirdly, we consider the node prestige to rank the answers, while BLINKS does not. This proposal is also orthogonal to our previous study EASE [44]. Firstly, EASE identifies the Steiner graphs as the answers while this paper finds CSTrees to approximate the Steiner trees. Secondly, EASE is only efficient for undirected, unweighted graphs while this paper studies the directed/undirected weighted graphs. Thirdly, this paper incorporates the proposed methods into RDBMS and uses the capabilities of the RDBMS to facilitate keyword-based search. This paper is an extended version of our poster [45] by adding indexing, ranking, proofs, and extensive experimental evaluations.

In addition, many proposals [24, 65, 13, 46, 59, 41, 48, 66, 6, 36, 49, 56, 61] have been proposed to study the problem of keyword search over XML documents. They model an XML document as a tree structure and find minimal-cost subtree to answer keyword

queries. Typically they compute lowest common ancestors (LCAs) or their variants to answer keyword queries. As an extension to LCA, XRank [24], XSearch [13], meaningful LCA (MLCA) [46], smallest LCA (SLCA) [65], multiway-SLCA [59], valuable LCA (VLCA) [41], XSeek [48], RACE [40,18,43], and exclusive LCA (ELCA) [66] have been recently proposed to improve search efficiency and effectiveness. They focus on proposing different semantics and various algorithms. XKeyword [29] is a system that offers keyword proximity search over XML documents. However, it models XML documents as graphs by considering IDREFs. Hristidis et al. [27] approached the problem of answering keyword queries over XML documents using grouped distance minimum connecting trees. The problem of effective keyword search over XML views has recently been investigated by Shao et al [57]. Our method can be extended to support keyword search in XML data. An XML document usually can be modeled as a tree structure. Given a keyword query $\{k_i, k_2, \cdots, k_m\}$, for each node n in the XML tree, we generate the CSTree rooted node n, which contains the path from node n to n's descendants that contain a keyword k_i and have the minimum distance to n. In this way, we can use the same idea to answer keyword queries in XML data.

9 Conclusion

We have studied the problem of effective keyword search over relational databases by identifying a more compact and meaningful type of Steiner trees. The minimum path weight Steiner trees have been devised as an efficient and effective approximate solution to the minimum Steiner tree problem. To make this approximate solution work for *top-k* queries, a novel concept of CSTree is proposed. CSTrees can be effectively identified in decreasing order of their individual ranks, while minimum path weight Steiner trees can be easily reconstructed from CSTrees. Furthermore, to improve search efficiency and result quality, we have designed a novel structure-aware index by materializing the structural relationships and ranking scores into the index. This novel indexing method can be seamlessly incorporated into any existing RDBMS, without the need to modify the source code of RDBMS. It can achieve a good performance by using the capabilities of the underlying RDBMS to support keyword-based search in relational databases. Our proposed approach has been implemented in MYSQL. The experimental results confirm that our approach can achieve high efficiency and result quality, and significantly outperforms state-of-the-art methods.

10 Acknowledgment

This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Program of China under Grant No.2007AA01Z152 & 2009AA011906, the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303103, and Australia Research Council (ARC) Grants DP0987557 and LP0882957.

References

1. S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

2. S. Amer-Yahia, D. Hiemstra, T. Roelleke, D. Srivastava, and G. Weikum. Db&ir integration: report on the dagstuhl seminar "ranked xml querying". *SIGMOD Record*, 37(3):46–49, 2008.
3. B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top- algorithms on exact and fuzzy data sets. *VLDB J.*, 18(2):407–427, 2009.
4. F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.
5. A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.
6. Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.
7. H. Bast and I. Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, pages 88–95, 2007.
8. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.
9. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
10. S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.
11. Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.
12. E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD Conference*, pages 349–360, 2009.
13. S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.
14. B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. In *VLDB*, pages 1189–1204, 2008.
15. 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.
16. R. Fagin. Fuzzy queries in multimedia database systems. In *PODS*, pages 1–10, 1998.
17. I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
18. J. Feng, G. Li, J. Wang, and L. Zhou. Finding and ranking compact connected trees for effective keyword proximity search in xml documents. *Information Systems*, 2009.
19. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
20. M. R. Garey and D. S. Johnson. The rectilinear steiner tree problem in np complete. *SIAM Journal of Applied Mathematics*, 32:826–834, 1977.
21. N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *J. Algorithms*, 37(1):66–84, 2000.
22. K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940, 2008.
23. L. Guo, J. Shanmugasundaram, and G. Yona. Topology search over biological databases. In *ICDE*, pages 556–565, 2007.
24. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.
25. H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.
26. V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.
27. V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. In *IEEE TKDE 18(4)*, pages 525–539, 2006.
28. V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.
29. V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.
30. M. Hua, J. Pei, A. W.-C. Fu, X. Lin, and H.-F. Leung. Top-k typicality queries and efficient query answering methods on large databases. *VLDB J.*, 18(3):809–835, 2009.
31. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

32. S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.
33. 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, 2005.
34. B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.
35. J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.
36. L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, pages 815–826, 2009.
37. G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. In *EDBT*, pages 391–402, 2009.
38. R. Lempel and S. Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst.*, 19(2):131–160, 2001.
39. G. Li, J. Feng, J. Wang, X. Song, and L. Zhou. Sailer: an effective search engine for unified retrieval of heterogeneous xml and web documents. In *WWW*, pages 1061–1062, 2008.
40. G. Li, J. Feng, J. Wang, B. Yu, and Y. He. Race: finding and ranking compact connected trees for keyword proximity search over xml documents. In *WWW*, pages 1045–1046, 2008.
41. G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.
42. G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.
43. G. Li, C. Li, J. Feng, and L. Zhou. Sail: Structure-aware indexing for effective and progressive top-k keyword search over xml documents. *Information Sciences*, 2009.
44. 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 Conference*, pages 903–914, 2008.
45. G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *ICDE*, 2009.
46. Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.
47. F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.
48. Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.
49. Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.
50. Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.
51. A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD Conference*, pages 605–616, 2007.
52. L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of rdbms. In *SIGMOD Conference*, pages 681–694, 2009.
53. M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *NIPS*, pages 1441–1448, 2001.
54. G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. In *SODA*, pages 770–779, 2000.
55. M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, pages 346–355, 2007.
56. F. Shao, L. Guo, C. Botev, A. Bhaskar, M. Chettiar, F. Y. 0002, and J. Shanmugasundaram. Efficient keyword search over virtual xml views. *VLDB J.*, 18(2):543–570, 2009.
57. F. Shao, L. Guo, C. Botev, A. Bhaskar, M. M. M. Chettiar, F. Yang, and J. Shanmugasundaram. Efficient keyword search over virtual xml views. In *VLDB*, pages 1057–1068, 2007.
58. A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.
59. C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.
60. Y. Tao and J. X. Yu. Finding frequent co-occurring terms in relational keyword search. In *EDBT*, pages 839–850, 2009.

61. M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top- k query processing for semistructured data. *VLDB J.*, 17(1):81–115, 2008.
62. T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, pages 405–416, 2009.
63. Q. H. Vu, B. C. Ooi, D. Papadias, and A. K. H. Tung. A graph method for keyword-based selection of the top-k databases. In *SIGMOD Conference*, pages 915–926, 2008.
64. G. Weikum. Db&ir: both sides now. In *SIGMOD Conference*, pages 25–30, 2007.
65. Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.
66. Y. Xu and Y. Papakonstantinou. Efficient LCA based keyword search in XML data. In *EDBT*, pages 535–546, 2008.
67. B. Yu, G. Li, K. R. Sollins, and A. K. H. Tung. Effective keyword-based selection of relational databases. In *SIGMOD Conference*, pages 139–150, 2007.
68. D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.