Keyword Search in Relational Databases: A Survey

Jeffrey Xu Yu, Lu Qin, Lijun Chang
Chinese University of Hong Kong
{yu,lqin,ljchang}@se.cuhk.edu.hk

1 Introduction

The integration of DB and IR provides flexible ways for users to query information in the same platform [2, 3, 5–7, 28]. On one hand, the sophisticated DB facilities provided by RDBMSs assist users to query well-structured information using SQL. On the other hand, IR techniques allow users to search unstructured information using keywords based on scoring and ranking, and do not need users to understand any database schemas.

We survey the developments on finding *structural information* among tuples in an *RDB* using an l-keyword query, Q, which is a set of keywords of size l, denoted as $Q = \{k_1, k_2, \cdots, k_l\}$. Here, an *RDB* is viewed as a data graph $G_D(V, E)$, where V represents a set of tuples, and E represents a set of edges between tuples. An edge exists between two tuples if at least there is a foreign key reference from one to the other. A tuple consists of attribute values and some of them are strings or full-text. The structural information to be returned for an l-keyword query is a set of connected structures, $\mathcal{R}$, where a connected structure represents how the tuples, that contain the required keywords, are interconnected in a database G_D. $\mathcal{R}$ can be either all trees or all subgraphs. When a function $score(\cdot)$ is given to score a structure, we can find the top-k structures instead of all structures in G_D. Such a $score(\cdot)$ function can be based on either the text information maintained in tuples (node weights), or the connections among tuples (edge weights), or both.

In Section 2, we focus on supporting keyword search in an RDBMS using SQL. Since this implies making use of the database schema information to issue SQL queries in order to find structures for an l-keyword query, it is called the schema-based approach. The two main steps in the schema-based approach are how to generate a set of SQL queries that can find all the structures among tuples in an *RDB* completely, and how to evaluate the generated set of SQL queries efficiently. Due to the nature of set operations used in SQL and the underneath relational algebra, a data graph G_D is considered as an undirected graph by ignoring the direction of references between tuples, and therefore a returned structure is of undirected structure (either tree or subgraph). The existing algorithms use a parameter to control the maximum size of a structure allowed. Such a size control parameter limits the number of SQL queries to be executed. Otherwise, the number of SQL queries to be executed for finding all or even top-k structures is too large. The $score(\cdot)$ functions used to rank the structures are all based on the text information on tuples.

In Section 3, we focus on supporting keyword search in an RDBMS from a different viewpoint, by materializing an *RDB* as a directed graph G_D. Unlike an undirected graph, the fact that a tuple v can reach to another tuple u in a directed graph does not necessarily mean that the tuple v is reachable from u. In this context, a returned structure (either steiner tree, distinct rooted tree, r-radius steiner graph, or multi-center subgraph) is directed. Such direction handling provides users with more information on how the tuples are interconnected.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

On the other hand, it requests higher computational cost to find such structures. Many graph-based algorithms are designed to find top-k structures, where the $score(\cdot)$ functions used to rank the structures are mainly based on the connections among tuples. This type of approach is called schema-free in the sense that it does not request any database schema assistance.

2 Schema-Based Keyword Search on Relational Databases

Consider a relational database schema as a directed graph $G_S(V, E)$, called a *schema graph*, where V represents the set of relation schemas $\{R_1, R_2, \cdots, R_n\}$ and E represents the set of edges between two relation schemas. Given two relation schemas, R_i and R_j, there exists an edge in the schema graph, from R_i to R_j, denoted $R_i \rightarrow R_j$, if the primary key defined on R_i is referenced by the foreign key defined on R_j. There may exist multiple edges from R_i to R_j in G_S if there are different foreign keys defined on R_j referencing the primary key defined on R_i. In such a case, $R_i \stackrel{X}{\rightarrow} R_j$ is used, where X is the foreign key attribute names. We use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of a graph G, respectively. In a relation schema R_i, we call an attribute, defined on strings or full-text, a text attribute, to which keyword search is allowed. A relation on relation schema R_i is an instance of the relation schema (a set of tuples) conforming to the relation schema, denoted $r(R_i)$. We use R_i to denote $r(R_i)$ if the context is obvious. A relational database (*RDB*) is a collection of relations. An *RDB* can be viewed as a data graph $G_D(V, E)$ on the schema graph G_S. Here, $V(G_D)$ represents a set of tuples, and $E(G_D)$ represents a set of edges between tuples. There is an edge between two tuples t_i and t_j in G_D, if there exists a foreign key reference from t_i to t_j or vice versa (undirected) in the *RDB*. In general, two tuples, t_i and t_j are reachable if there exists a sequence of connections between t_i and t_j in G_D. The distance $dist(t_i, t_j)$ between two tuples t_i and t_j is defined as the minimum number of connections between t_i and t_j.

An l-keyword query is given as a set of keywords of size l, $Q = \{k_1, k_2, \cdots, k_l\}$, and searches interconnected tuples that contain the given keywords, where a tuple contains a keyword if a text attribute of the tuple contains the keyword. To select all tuples from a relation R that contain a keyword k_1, a predicate $contain(A, k_1)$ is supported in SQL in IBM *DB2*, *ORACLE*, and Microsoft SQL-SERVER, where A is a text attribute in R. The following SQL query, finds all tuples in R containing k_1 provided that the attributes A_1 and A_2 are all and the only text attributes in relation R. We say a tuple contains a keyword, for example k_1, if the tuple is included in the result of such a selection.

select * **from** R **where** $contain(A_1, k_1)$ **or** $contain(A_2, k_1)$

An l-keyword query returns a set of answers, where an answer is a minimal total joining network of tuples (*MTJNT*) [1, 16] that is defined as follows. Given an l-keyword query and a relational database with schema graph G_S, a joining network of tuples (*JNT*) is a connected tree of tuples where every two adjacent tuples, $t_i \in r(R_i)$ and $t_j \in r(R_j)$ can be joined based on the foreign key reference defined on relational schema R_i and R_j in G_S (either $R_i \rightarrow R_j$ or $R_j \rightarrow R_i$). An *MTJNT* is a joining network of tuples that satisfy the following two conditions, total and minimal. By total, each keyword in the query must be contained in at least one tuple of the joining network. By minimal, a joining network of tuples is not total if any tuple is removed.

Because it is meaningless if two tuples in an *MTJNT* are too far away from each other, a size control parameter, Tmax, is introduced to specify the maximum number of tuples allowed in an *MTJNT*.

Given an *RDB* on the schema graph G_S, in order to generate all the *MTJNT*s for an l-keyword query, Q, *keyword relation* and *Candidate Network* (*CN*) are defined as follows. A keyword relation $R_i\{K'\}$ is a subset of relation R_i containing tuples that only contain keywords $K'(\subseteq Q)$) and no other keywords, as defined below:

$$R_i\{K'\} = \{t | t \in r(R_i) \wedge \forall k \in K', t \text{ contains } k \wedge \forall k \in (K - K'), t \text{ does not contain } k\}$$

where K is the set of keywords in Q, i.e. $K = Q$. K' can be $\emptyset$. In such a situation, $R_i\{\}$ consists of tuples that do not contain any keywords in Q and is called an empty keyword relation. A candidate network (*CN*) is a connected tree of keyword relations where for every two adjacent keyword relations $R_i\{K_1\}$ and $R_j\{K_2\}$, we have $(R_i, R_j) \in E(G_S)$ or $(R_j, R_i) \in E(G_S)$. A candidate network must satisfy the following two conditions, total and minimal. By total, each keyword in the query must be contained in at least one keyword relation of the candidate network. By minimal, a candidate network is not total if any keyword relation is removed.

A *CN* can produce a set of (possibly empty) *MTJNT*s, and it corresponds to a relational algebra that joins a sequence of relations to obtain *MTJNT*s over the relations involved. Given a keyword query Q and an *RDB* with schema graph G_S, let $\mathcal{C} = \{C_1, C_2, \cdots\}$ be the set of all candidate networks for Q over G_S, and let $\mathcal{T} = \{T_1, T_2, \cdots\}$ be the set of all *MTJNT*s for Q over the *RDB*. For every $T_i \in \mathcal{T}$, there is exactly one $C_j \in \mathcal{C}$ that produces T_i.

For an l-keyword query over an *RDB*, the number of *MTJNT*s can be very large even if Tmax is small. It is ineffective to present users a huge number of results for a keyword query. In order to handle the effectiveness, for each *MTJNT*, T, for a keyword query Q, it also allows a score function $score(T, Q)$ defined on T in order to rank results. A top-k keyword query retrieves k *MTJNT*s $\mathcal{T} = \{T_1, T_2, ..., T_k\}$ such that for any two *MTJNT*s T and T' where $T \in \mathcal{T}$ and $T' \notin \mathcal{T}$, $score(T, Q) \leq score(T', Q)$.

Ranking issues for *MTJNT*s are discussed in many papers [14, 22, 23]. They aim at designing effective ranking functions that capture both the textual information (e.g., IR-Styled ranking) and structural information (e.g., the size of the *MTJNT*) for an *MTJNT*. There are two categories of ranking functions, namely, the attribute level ranking function and the tree level ranking function. Given an *MTJNT* T and a keyword query Q, the attribute level ranking function first assigns each text attribute for tuples in T an individual score and then combines them together to get the final score [14, 22]. In other words, in the attribute level ranking functions, each text attribute of an *MTJNT* is considered as a virtual document. Tree level ranking functions consider the whole *MTJNT* as a virtual document rather than each individual text attribute [23].

In the framework of RDBMS, the two main steps of processing an l-keyword query are candidate network generation and candidate network evaluation.

1. **Candidate Network Generation:** In the candidate network generation step, a set of candidate networks $\mathcal{C} = \{C_1, C_2, \cdots\}$ is generated over a graph schema G_S. The set of *CN*s shall be complete and duplication-free. The former ensures that all *MTJNT*s are found, and the latter is mainly for efficiency consideration.
2. **Candidate Network Evaluation:** In the candidate network evaluation step, all $C_i \in \mathcal{C}$ are evaluated.

We will introduce the two steps one by one in the next two sections.

2.1 Candidate Network Generation

In order to generate all candidate networks for an l-keyword query Q over an *RDB* with schema graph G_S, algorithms are designed to generate candidate networks $\mathcal{C} = \{C_1, C_2, ...\}$ that satisfy the following two conditions:

- **Complete:** For each solution T of the keyword query, there exists a candidate network $C_i \in \mathcal{C}$ that can produce T.
- **Duplication-Free:** For every two *CN*s $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$, C_i and C_j are not isomorphic to each other.

The complete and duplication-free conditions ensure that (1) all results (*MTJNT*s) for a keyword query will be produced by the set of *CN*s generated (due to completeness); and (2) any result T for a keyword query will be produced only once, i.e., there does not exist two *CN*s $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$ such that C_i and C_j both produce T (due to the duplication-free condition).

The first algorithm to generate all *CN*s was proposed in *DISCOVER* [16]. It expands the partial *CN*s generated to larger partial *CN*s until all *CN*s are generated. As the number of partial *CN*s can be exponentially large,

arbitrarily expanding will make the algorithm extremely inefficient. In *DISCOVER* [16], there are three pruning rules for partial *CN*s.

- **Rule-1:** Duplicated *CN*s are pruned (based on tree isomorphism).
- **Rule-2:** A *CN* can be pruned if it contains all the keywords and there is a leaf node, $R_j\{K'\}$, where $K' = \emptyset$, because it will generate results that do not satisfy the condition of minimality.
- **Rule-3:** When there only exists a single foreign key reference between two relation schemas (for example, $R_i \rightarrow R_j$), *CN*s including $R_i\{K_1\} \rightarrow R_j\{K_2\} \leftarrow R_i\{K_3\}$ will be pruned, where K_1, K_2, and K_3 are three subsets of Q, and $R_i\{K_1\}$, $R_j\{K_2\}$, and $R_i\{K_3\}$ are keyword relations.

The Rule-3 reflects the fact that the primary key defined on R_i and a tuple in the relation of $R_j\{K_2\}$ must refer to the same tuple appearing in both relations $R_i\{K_1\}$ and $R_i\{K_3\}$. As the same tuple cannot appear in two sub-relations in a *CN* (otherwise, it will not produce a valid *MTJNT* because the minimal condition will not be satisfied), the join results for $R_i\{K_1\} \rightarrow R_j\{K_2\} \leftarrow R_i\{K_3\}$ will not contain any valid *MTJNT*.

The algorithm in [16] can generate a complete and duplication-free set of *CN*s, but the cost of generating the set of *CN*s is high. *S-KWS* [24] proposes an algorithm (1) to reduce the number of partial results generated by expanding from part of the nodes in a partial tree and (2) to avoid isomorphism testing by assigning a proper expansion order.

2.2 Candidate Network Evaluation

After generating all candidate networks (*CN*s) in the first phase, the second phase is to evaluate all candidate networks in order to get the final results. *DBXplorer* [1], *DISCOVER* [16], *S-KWS* [24], *KDynamic* [27] and *KRDBMS* [25] compute all *MTJNT*s upon the set of *CN*s generated by specifying a proper execution plan. *DISCOVER-II* [14] and *SPARK* [23] compute top-k *MTJNT*s.

2.2.1 Getting all *MTJNT*s in a relational database

In RDBMS, the problem of evaluating all *CN*s in order to get all *MTJNT*s is a multi-query optimization problem. There are two main issues: (1) How to share common subexpressions among *CN*s generated in order to reduce computational cost when evaluating. (2) How to find a proper join order to fast evaluate all *CN*s. For a keyword query, the number of *CN*s generated can be very large. Given a large number of joins, it is extremely difficult to obtain an optimal query processing plan, because one best plan for a *CN* may slow down others, if its subtrees are shared by other *CN*s. As studied in *DISCOVER* [16], finding the optimal execution plan is an NP-complete problem.

In *DISCOVER* [16], an algorithm is proposed to evaluate all *CN*s together using a greedy algorithm based on the following observations: (1) subexpressions that are shared by most *CN*s should be evaluated first; and (2) subexpressions that may generate the smallest number of results should be evaluated first. In *S-KWS* [24], in order to share the computational cost of evaluating all *CN*s, Markowetz et al. construct an operator mesh. In a mesh, there are $n \cdot 2^{l-1}$ clusters, where n is the number of relations in the schema graph G_S and l is the number of keywords. A cluster consists of a set of operator trees (left-deep trees) that share common expressions. When evaluating all *CN*s in a mesh, a projected relation with the smallest number of tuples is selected to start and to join. In *KDynamic* [27], a $\mathcal{L}$-Lattice is introduced to share computational cost among *CN*s. Given a set of *CN*s, $\mathcal{C}$, it defines the root of each *CN* to be the node r such that the maximum length of the path from r to all leaf nodes of the *CN* is minimized. There are three main differences between the Mesh and the $\mathcal{L}$-Lattice. (1) The maximum depth of a Mesh is $\mathsf{Tmax} - 1$ and the maximum depth of an $\mathcal{L}$-Lattice is $\lfloor \mathsf{Tmax}/2 + 1 \rfloor$. (2) In a mesh, only the left part of two *CN*s can be shared (except for the leaf nodes), while in an $\mathcal{L}$-Lattice multiple parts of two *CN*s can be shared. (3) The number of leaf nodes in a mesh is $O((|V(G_S)| \cdot 2^l)^2)$ because there

are $O(|V(G_S)| \cdot 2^l)$ clusters in a mesh and each cluster may contain $O(|V(G_S)| \cdot 2^l)$ leaf nodes. The number of leaf nodes in an $\mathcal{L}$-Lattice is $O(2^l)$.

After sharing computational cost using either the Mesh or the $\mathcal{L}$-Lattice, all *CN*s are evaluated using joins in *DISCOVER* or *S-KWS*. In *KRDBMS* [25], the authors observe that evaluating all *CN*s using only joins may always generate a large number of temporary tuples. They propose to use semijoin/join sequences to evaluate a *CN*.

Besides evaluating all *CN*s in a static environment, *S-KWS* and *KDynamic* focus on monitoring all *MTJNT*s in a relational data stream, where tuples can be inserted/deleted frequently. In this situation, it is necessary to find new *MTJNT*s or expire *MTJNT*s in order to monitor events that are implicitly interrelated over an open-ended relational data stream for a user-given l-keyword query. In other words, it reports new *MTJNT*s when new tuples are inserted, and, in addition, reports the *MTJNT*s that become invalid when tuples are deleted. A sliding window (time interval), W, is specified. A tuple, t, has a lifespan from its insertion into the window at time $t.start$ to $W + t.start - 1$, if t is not deleted before then. Two tuples can be joined if their lifespans overlap.

2.2.2 Getting top-k *MTJNT*s in a relational database

A naive approach to answer the top-k keyword queries is to first generate all *MTJNT*s using the algorithms proposed in Section 2.2.1, and then calculate the score for each *MTJNT*, and finally output the top-k *MTJNT*s with the highest scores. In *DISCOVER-II* [14] and *SPARK* [23], several algorithms are proposed to get top-k *MTJNT*s efficiently. The aim of all the algorithms is to find a proper order of generating *MTJNT*s in order to stop early before all *MTJNT*s are generated.

In *DISCOVER-II*, three algorithms are proposed to get top-k *MTJNT*s, namely, the Sparse algorithm, the Single-Pipelined algorithm, and the Global-Pipelined algorithm. All algorithms are based on the attribute level ranking function, which has the property of *tuple monotonicity*, defined as follows. For any two *MTJNT*s $T = t_1 \bowtie t_2 \bowtie ... \bowtie t_l$ and $T' = t'_1 \bowtie t'_2 \bowtie ... \bowtie t'_l$ generated from the same *CN* C, if for any $1 \leq i \leq l$, $score(t_i, Q) \leq score(t'_i, Q)$, then we have $score(T, Q) \leq score(T', Q)$. With the tuple monotonicity, the algorithms are designed to stop early where possible.

In *SPARK* [23], the authors study the tree level ranking function which does not satisfy tuple monotonicity. In order to handle such non-monotonic score functions, a new monotonic upper bound function is introduced. The intuition behind the upper bound function is that, if the upper bound score is already smaller than the score of a certain result, then all the upper bound scores of unseen tuples will be smaller than the score of this result due to the monotonicity of the upper bound function. Two new algorithms are proposed in *SPARK*: Skyline-Sweeping and Block-Pipelined.

2.3 Other Keyword Search Semantics

In the above discussions, for an l-keyword query on an *RDB*, each result is an *MTJNT*. This is referred to as the *connected tree semantics*. There are two other semantics to answer an l-keyword query on an *RDB*, namely *distinct root semantics* and *distinct core semantics*.

Distinct Root Semantics: An l-keyword query finds a collection of tuples that contain all the keywords and that are reachable from a root tuple (center) within a user-given distance (Dmax). The distinct root semantics implies that the same root tuple determines the tuples uniquely [9, 13, 15, 21, 25]. Suppose that there is a result rooted at tuple t_r. For any of the l keywords, say k_i, there is a tuple t in the result that satisfies the following conditions: (1) t contains keyword k_i, (2) among all tuples that contain k_i, the distance between t and t_r is minimum[1], and (3) the minimum distance between t and t_r must be less than or equal to a user given parameter Dmax.

[1] If there is a tie, then a tuple is selected with a predefined order among tuples in practice.

Distinct Core Semantics: An l-keyword query finds multi-center subgraphs, called communities [25]. A community, $C_i(V, E)$, is specified as follows. V is a union of three subsets of tuples, $V = V_c \cup V_k \cup V_p$, where, V_k is a set of keyword-tuples where a keyword-tuple $v_k \in V_k$ contains at least a keyword and all l keywords in the given l-keyword query must appear in at least one keyword-tuple in V_k; V_c is a set of center-tuples where there exists at least a sequence of connections between $v_c \in V_c$ and every $v_k \in V_k$ such that $dist(v_c, v_k) \leq \mathsf{Dmax}$; and V_p is a set of path-tuples that appear on a shortest sequence of connections from a center-tuple $v_c \in V_c$ to a keyword-tuple $v_k \in V_k$ if $dist(v_c, v_k) \leq \mathsf{Dmax}$. Note that a tuple may serve several roles as keyword/center/path tuples in a community. E is a set of connections for every pair of tuples in V if they are connected over shortest paths from nodes in V_c to nodes in V_k. A community, C_i, is uniquely determined by the set of keyword tuples, V_k, which is called the core of the community, and denoted as $\mathsf{core}(C_i)$.

In [25], the authors showed tuple-reduction approaches to process different semantics using SQL in RDBMSs.

3 Graph-Based Keyword Search

A data graph G_D can be considered as materialization of an *RDB*. In this section, we show how to answer keyword queries using graph algorithms. We consider a weighted directed graph in this section, $G_D(V, E)$. Weights are assigned to edges to reflect the (directional) proximity of the corresponding tuples, denoted as $w_e(\langle u, v\rangle)$. A commonly used weighting scheme [4, 10] is as follows. For a foreign key reference from t_u to t_v, the weight for the directed edge $\langle u, v\rangle$ is given as Eq. 4, and the weight for the backward edge $\langle v, u\rangle$ is given as Eq. 5.

$$w_e(\langle u, v\rangle) = 1 \quad (4)$$

$$w_e(\langle v, u\rangle) = \log_2(1 + N_{in}(v)) \quad (5)$$

where $N_{in}(v)$ is the number of tuples that refer to t_v, which is the tuple corresponding to node v. Nodes can have weights. But, because the algorithms that deal with edge-weighted graphs can be easily modified to handle additional node-weights, below we assume that only edges have weights. We denote the number of nodes and the number of edges in graph, G_D, using $n = |V(G_D)|$ and $m = |E(G_D)|$.

There are different structures of tuples to be returned: (1) a reduced tree that contains all the keywords, that we refer to as *tree-based semantics*; (2) a subgraph, such as r-radius steiner graph [21], and multi-center induced graph [26], we call this *subgraph-based semantics*. In the following, we focus the tree-based semantics, and we will discuss the subgraph-based semantics in Section 3.3.

In the tree-based semantics, an answer to Q (called a Q-SUBTREE) is defined as any subtree T of G_D that is reduced with respect to Q. Formally, there exists a sequence of l nodes in T, $\langle v_1, \cdots, v_l\rangle$ where $v_i \in V(T)$ and v_i contains keyword term k_i for $1 \leq i \leq l$, such that the leaves of T can only come from those nodes, i.e. $leaves(T) \subseteq \{v_1, v_2, \cdots, v_l\}$, the root of T should also be from those nodes if it has only one child, i.e. $root(T) \in \{v_1, v_2, \cdots, v_l\}$.

The Q-SUBTREE is popularly used to describe answers to keyword queries. Two different weight functions are proposed in the literature to rank Q-SUBTREEs in increasing weight order, and two semantics are proposed based on the two weight functions, namely *steiner tree-based semantics*, and *distinct root-based semantics*.

Steiner Tree-Based Semantics: In this semantics, the weight of a Q-SUBTREE is defined as the total weight of the edges in the tree; formally,

$$w(T) = \sum_{\langle u,v\rangle \in E(T)} w_e(\langle u, v\rangle) \quad (6)$$

where $E(T)$ is the set of edges in T. The l-keyword query finds all (or top-k) Q-SUBTREEs in weight increasing order, where the weight denotes the cost to connect the l keywords. Under this semantics, finding the Q-SUBTREE with the smallest weight is the well-known *optimal steiner tree problem* which is NP-complete [11].

Distinct Root-Based Semantics: Since the problem of keyword search under the steiner tree-based semantics is generally a hard problem, many works resort to easier semantics. Under the distinct root-based semantics, the weight of a Q-SUBTREE is the sum of the shortest distance from the root to each keyword node; more precisely,

$$w(T) = \sum_{i=1}^{l} dist(root(T), k_i) \tag{7}$$

where $root(T)$ is the root of T, $dist(root(T), k_i)$ is the shortest distance from the root to the keyword node k_i.

There are two differences between the two semantics. First is the weight function as shown above. The other difference is the total number of Q-SUBTREEs for a keyword query. In theory, there can be exponentially many Q-SUBTREEs under the steiner tree semantics, i.e., $O(2^m)$ where m is the number of edges in G_D. But, under the distinct root semantics, there can be at most n, which is the number of nodes in G_D, Q-SUBTREEs, i.e. zero or one Q-SUBTREE rooted at each node $v \in V(G_D)$. The potential Q-SUBTREE rooted at v is the union of the shortest path from v to each keyword node k_i.

3.1 Steiner Tree-Based Keyword Search

In this section, we show three categories of algorithms under the steiner tree-based semantics. First is the backward search algorithm, where the first tree returned is an l-approximation of the optimal steiner tree. Second is a dynamic programming approach, which finds the optimal (top-1) steiner tree in time $O(3^l n + 2^l((l + \log n)n + m))$. Third is enumeration algorithms with polynomial delay.

3.1.1 Backward Search

BANKS-I [4] enumerates Q-SUBTREEs using a backward search algorithm searching backwards from the nodes that contain keywords. Given a set of l keywords, they first find the set of nodes that contain keywords, S_i, for each keyword term k_i, i.e. S_i is exactly the set of nodes in $V(G_D)$ that contain the keyword term k_i. This step can be accomplished efficiently using an inverted list index. Let $\mathcal{S} = \bigcup_{i=1}^{l} S_i$. Then, the backward search algorithm concurrently runs $|\mathcal{S}|$ copies of Dijkstra's single source shortest path algorithm, one for each keyword node v in $\mathcal{S}$ with node v as the source. The $|\mathcal{S}|$ copies of Dijkstra's algorithm run concurrently using iterators. All the Dijkstra's single source shortest path algorithms traverse graph G_D in reverse direction. When an iterator for keyword node v visits a node u, it finds a shortest path from u to the keyword node v. The idea of concurrent backward search is to find a common node from which there exists a shortest path to at least one node in each set S_i. Such paths will define a rooted directed tree with the common node as the root and the corresponding keyword nodes as the leaves.

The connected trees computed by BANKS-I are approximately sorted in increasing weight order. Computing all the connected trees followed by sorting would increase the computation time and also lead to a greatly increased time to output the first result. A fixed-size heap is maintained as a buffer for the computed connected trees. Newly computed trees are added into the heap. Whenever the heap is full, the top result tree is output and removed. With BANKS-I, the first Q-SUBTREE output is an l-approximation of the optimal steiner tree, and the Q-SUBTREEs are computed in increasing height order. The Q-SUBTREEs computed by BANKS-I is not complete, as BANKS-I only considers the shortest path from the root of a tree to nodes containing keywords.

3.1.2 Dynamic Programming

Although finding the optimal steiner tree (top-1 Q-SUBTREE under the steiner tree-based semantics) or group steiner tree is NP-complete in general, there are efficient algorithms to find the optimal steiner tree for l-keyword queries [10,19], because l is small. The algorithm [10] solves the group steiner tree problem. Note that the group steiner tree in a directed (or undirected) graph can be transformed into steiner tree problem in directed graph.

Let $\mathbf{k}, \mathbf{k1}, \mathbf{k2}$ denote a non-empty subset of the keyword nodes $\{k_1, \cdots, k_l\}$. Let $T(v, \mathbf{k})$ denote the tree with the minimum weight (called it *optimal tree*) among all the trees rooted at v and containing all the keyword nodes in $\mathbf{k}$. We can find the optimal tree $T(v, \mathbf{k})$ for each $v \in V(G_D)$ and $\mathbf{k} \subseteq Q$. Initially, for each keyword node k_i, $T(k_i, \{k_i\})$ is a single node tree consisting of the keyword node k_i with tree weight 0. For a general case, the $T(v, \mathbf{k})$ can be computed by the following equations.

$$T(v, \mathbf{k}) = \min(T_g(v, \mathbf{k}), T_m(v, \mathbf{k})) \quad (8)$$

$$T_g(v, \mathbf{k}) = \min_{\langle v,u\rangle \in E(G_D)} \{\langle v, u\rangle \oplus T(u, \mathbf{k})\} \quad (9)$$

$$T_m(g, \mathbf{k1} \cup \mathbf{k2}) = \min_{\mathbf{k1} \cap \mathbf{k2} = \emptyset} \{T(v, \mathbf{k1}) \oplus T(v, \mathbf{k2})\} \quad (10)$$

Here, $\min$ means to choose the tree with minimum weight from all the trees in the argument. Note that, $T(v, \mathbf{k})$ may not exist for some v and $\mathbf{k}$, which reflects that node v can not reach some of the keyword nodes in $\mathbf{k}$, then $T(v, \mathbf{k}) = \perp$ with weight ∞. $T_g(v, \mathbf{k})$ reflects the tree grow case, and $T_m(v, \mathbf{k})$ reflects the tree merge case. An algorithm called DPBF for Best-First Dynamic Programming is proposed in [10]. Consider a graph G_D with n nodes and m edges, DPBF finds the optimal steiner three containing all the keywords in $Q = \{k_1, \cdots, k_l\}$, in time $O(3^l n + 2^l((l + n) \log n + m))$ [10].

DPBF can be modified slightly to output k steiner trees in increasing weight order, denoted as DPBF-k, by terminating DPBF after finding k steiner trees that contain all the keywords.

3.1.3 Enumerating Q-SUBTREEs with Polynomial Delay

Although BANKS-I can find an l-approximation of the optimal Q-SUBTREE and DPBF can find the optimal Q-SUBTREE, the non-first results returned by these algorithms can not guarantee their quality (or approximation ratio), and the delay between consecutive results can be very large. In [12, 18], the authors show three algorithms to enumerate Q-SUBTREEs in increasing (or θ-approximate increasing) weight order with polynomial delay: (1) an enumeration algorithm enumerates Q-SUBTREEs in increasing *weight* order with polynomial delay under the *data complexity*, (2) an enumeration algorithm enumerates Q-SUBTREEs in $(\theta + 1)$-approximate *weight* order with polynomial delay under *data-and-query complexity*, (3) an enumeration algorithm enumerates Q-SUBTREEs in 2-approximate *height* order with polynomial delay under *data-and-query complexity*. The algorithms are adaption of the Lawler's procedure [20] to enumerate Q-SUBTREEs in rank order.

3.2 Distinct Root-Based Keyword Search

In this section, we show approaches to find Q-SUBTREEs using the distinct root semantics, where the weight of a tree is defined as the sum of the shortest distance from the root to each keyword node. As shown in the previous section, the problem of keyword search under the directed steiner tree is, in general, a hard problem. Using the distinct root semantics, there can be at most n Q-SUBTREEs for a keyword query, and in the worst case, all the Q-SUBTREEs can be found in time $O(l(n \log n + m))$. The approaches introduced in this section deal with very large graphs in general, and they propose search strategies or indexing schemes to reduce the search time for an online keyword query.

3.2.1 Bidirectional Search

BANKS-I algorithm can be directly applied to the distinct root semantics. It would explore an unnecessarily large number of nodes in the following two scenarios. First, the query contains a frequently occurring keyword. In BANKS-I, one iterator is associated with each keyword node. The algorithm would generate a large number of iterators if a keyword matches a large number of nodes. Second, an iterator reaches a node with large fan-in

(incoming edges). An iterator may need to explore a large number of nodes if it hits a node with a very large fan-in. BANKS-II [17] is proposed to overcome the drawbacks of BANKS-I. The main idea of bidirectional search is to start forward searches from potential roots. The main differences of bidirectional search from BANKS-I are as follows. First, all the single source shortest path iterators from the BANKS-I algorithm are merged into a single iterator, called the *incoming iterator*. Second, an *outgoing iterator* runs concurrently, which follows forwarding edges starting from all the nodes explored by the *incoming iterator*. Third, *spreading activation* is proposed to prioritize the search, which chooses *incoming iterator* or *outgoing iterator* to be called next. It also chooses the next node to be visited in the *incoming iterator* or *outgoing iterator*.

3.2.2 Bi-level Indexing

BLINKS [13] is proposed as a bi-level index to speed up BANKS-II, as no index (except the keyword-node index) is used in BANKS-II. A naive index precomputes and indexes all the distances from the nodes to keywords, but this will incur very large index size, as the number of distinct keywords is in the order of the size of the data graph G_D. A bi-level index can be built by first partitioning graph, and then building intra-block index and block index. Two node-based partitioning methods are proposed to partition a graph into blocks, namely, BFS-Based Partitioning, and METIS-Based Partitioning.

In [13], in a node-based partitioning of a graph, a node separator is called a *portal* node (or portal). A block consists of all nodes in a partition as well as all portals incident to the partition. For a block, a portal can be either "in-portal", "out-portal", or both. A portal is called in-portal if it has at least one incoming edge from another block and at least one outgoing edge in this block. And a portal is called out-portal if it has at least one outgoing edge to another block and at least one incoming edge from this block.

For each block b, the intra-block index (IB-index) is built.

In *BLINKS* [13], a priority queue $\mathcal{Q}_i$ of cursors is created for each keyword term k_i to simulate Dijkstra's algorithm by utilizing the distance information stored in the IB-index. Initially, for each keyword k_i, all the blocks that contain it are found by the keyword-block list, and a cursor is created to scan each intra-block keyword-node list and put in queue $\mathcal{Q}_i$. When an in-portal u is visited, all the blocks that have u as their out-portal need to be expanded, because a shorter path may cross several blocks.

3.2.3 External Memory Data Graph

Dalvi et al. study keyword search on graphs where the graph G_D can not fit into main memory [9]. They build a much smaller supernode graph on top of G_D that can resident in main memory. The supernode graph is defined as follows:

- **SuperNode:** The graph G_D is partitioned into components by a clustering algorithm, and each cluster is represented by a node called the *supernode* in the top-level graph. Each supernode thus contains a subset of $V(G_D)$, and the contained nodes (nodes in G_D) are called *innernodes*.
- **SuperEdge:** The edges between the supernodes called superedges are constructed as follows: if there is at least one edge from an innernode of supernode s_1 to an innernode of supernode s_2, then there exists a superedge from s_1 to s_2.

A supernode graph is constructed that fits into main memory, where each supernode has a fixed number of innernodes and is stored on disk.

A *multi-granular graph* is used to exploit information presented in lower-level nodes (innernodes) that are cache-resident at the time a query is executed. A multi-granular graph is a hybrid graph that contains both supernodes and innernodes. A supernode is present either in *expanded* form, i.e., all its innernodes along with their adjacency lists are present in the cache, or in *unexpanded* form, i.e., its innernodes are not in the cache.

The innernodes and their adjacency lists are handled in the unit of supernodes, i.e. either all or none of the innernodes of a supernode are presented in the cache.

When searching the multi-granular graph, the answers generated may contain supernodes, called *supernode answer*. If an answer does not contain any supernodes, it is called *pure answer*. The final answer returned to users must be pure answer. The Iterative Expansion Search algorithm (IES) [9] is a multi-stage algorithm that is applicable to multi-granular graphs. Each iteration of IES can be broken up into two phases.

- **Explore phase:** Run an in-memory search algorithm on the current state of the multi-granular graph. The multi-granular graph is entirely in memory, whereas the supernode graph is stored in main memory, and details of expanded supernodes are stored in cache. When the search reaches an expanded supernode, it searches on the corresponding innernodes in cache.
- **Expand phase:** Expand the supernodes found in top-n ($n > k$) results of the previous phase and add them to input graph to produce an expanded multi-granular graph, by loading all the corresponding innernodes into cache.

The graph produced at the end of Expand phase of iteration i acts as the graph for iteration $i + 1$. The algorithm stops when all top-k results are pure.

3.3 Subgraph-Based Keyword Search

The previous sections define the answer of a keyword query as Q-SUBTREE, which is a directed subtree. We show two subgraph-based notions of answer definition for a keyword query in the following, namely, r-radius steiner graph, and multi-center induced graph.

3.3.1 r-radius steiner graph

Li et al. in [21] define the result of an l-keyword query as an r-radius steiner subgraph. The graph is unweighted and undirected, and the length of a path is defined as the number of edges in it. The definition of r-radius steiner graph is based on *centric distance* and *radius*. The centric distance of v in G, denoted as $CD(v)$, is the maximum among the shortest distances between v and any node $u \in V(G)$, i.e. $CD(v) = \max_{u \in V(G)} dist(u, v)$. The radius of a graph G, denoted as $\mathcal{R}(G)$, is the minimum value among the centric distances of every node in G, i.e. $\mathcal{R}(G) = \min_{v \in V(G)} CD(v)$. G is called an r-radius graph if its radius is exactly r.

Given an r-radius graph G and a keyword query Q, node v in G is called a content node if it contains some of the input keywords. Node s is called steiner node if there exist two content nodes, u and v, and s in on the simple path between u and v. The subgraph of G composed of the steiner nodes and associated edges is called an r-radius steiner graph (SG). The radius of an r-radius steiner graph can be smaller than r.

The result of an l-keyword query, with a given radius, is a set of r-radius steiner graphs. The approaches to find r-radius steiner graphs are based on finding r-radius subgraphs using the adjacency matrix, $M = (m_{ij})_{n \times n}$, with respect to G_D, which is a $n \times n$ Boolean matrix. An element m_{ij} is 1, if and only if there is an edge between v_i and v_j, m_{ii} is 1 for all i. $M^r = M \times M \cdots \times M = (m_{ij})_{n \times n}$ is the r-th power of adjacency matrix M. An element m^r_{ij} is 1, if and only if the shortest path between v_i and v_j is less than or equal to r. $N^r_i = \{v_j | m^r_{ij} = 1\}$ is the set of nodes that have a path to v_i with distance no larger than r. G^r_i denotes the subgraph induced by the node set N^r_i. We use $G_i \trianglelefteq G_j$ to denote that G_i is a subgraph of G_j. The r-radius subgraph is defined based on G^r_i's. The following lemma is used to find all the r-radius subgraphs [21].

Lemma 1: [21] Given a graph G, with $\mathcal{R}(G) \geq r > 1$, $\forall i, 1 \leq i \leq |V(G)|$, G^r_i is an r-radius subgraph, if, $\forall v_k \in N^r_i$, $N^r_i \not\subseteq N^{r-1}_k$.

Note that, the above lemma is a sufficient condition for identifying r-radius subgraphs, but not a necessary condition. [21] only considers $n = |V(G)|$ subgraphs, each is uniquely determined by one node in G.

An r-radius subgraph G_i^r is maximal if and only if there is no other r-radius subgraph G_j^r that is a super graph of G_i^r, i.e. $G_i^r \trianglelefteq G_j^r$. [21] considers those maximal r-radius subgraphs G_i^r as the subgraphs that will generate r-radius steiner subgraphs. All these maximal r-radius subgraphs G_i^r are found, which can be pre-computed and indexed on the disk, because these maximal r-radius graph are query independent.

3.3.2 Multi-Center Induced Graph

In contrast to tree-based results that are single-center (root) induced trees, query answers can be multi-centered induced subgraphs of G_D. These are referred to as *communities* [26]. The nodes of a community $R(V, E)$, $V(R)$ is a union of three subsets, $V = V_c \cup V_l \cup V_p$, where V_l represents a set of keyword nodes (*knode*), V_c represents a set of center nodes (*cnode*) (for every *cnode* $v_c \in V_c$, there exists at least a single path such that $dist(v_c, v_l) \leq R_{max}$ for any $v_l \in V_l$, where R_{max} is introduced to control the size of a community), and V_p represents a set path nodes (*pnode*) that include all the nodes that appear on any path from a *cnode* $v_c \in V_c$ to a *knode* $v_l \in V_l$ with $dist(v_c, v_l) \leq R_{max}$. $E(R)$ is the set of edges induced by $V(R)$.

A community, R, is uniquely determined by the set of *knode*s, V_l, which is called the core of the community. The weight of a community R, $w(R)$ is defined as the minimum value among the total edge weights from a *cnode* to every *knode*; more precisely,

$$w(R) = \min_{v_c \in V_c} \sum_{v_l \in V_l} dist(v_c, v_l). \tag{11}$$

Algorithms are proposed in [26] to compute communities by adopting the Lawler's procedure [20].

4 Conclusion Remarks

In this article, we surveyed some main results on finding structural information in an *RDB* for an l-keyword query. The current work focus on identifying primitive structures as answers and efficiently computing all and/or top-k of such answers. One future work is how to compute more general structural information by making use of the primitive structures. More information can be found in [5, 7, 28].

Acknowledgment: We acknowledge the support of our research on keyword search by the grant of the Research Grants Council of the Hong Kong SAR, China, No. 419109.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. 18th Int. Conf. on Data Engineering*, pages 5–16, 2002.

[2] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the db/ir panel at sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[3] S. Amer-Yahia and J. Shanmugasundaram. Xml full-text search: Challenges and opportunities. In *Proc. 31st Int. Conf. on Very Large Data Bases*, page 1368, 2005.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. 18th Int. Conf. on Data Engineering*, pages 431–440, 2002.

[5] S. Chaudhuri and G. Das. Keyword querying and ranking in databases. *Proc. of the VLDB Endowment*, 2(2):1658–1659, 2009.

[6] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating db and ir technologies: What is the sound of one hand clapping? In *Proc. of CIDR'05*, 2005.

[7] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *Proc. 2009 ACM SIGMOD Int. Conf. On Management of Data*, pages 1005–1010, 2009.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[9] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *Proc. of the VLDB Endowment*, 1(1):1189–1204, 2008.

[10] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proc. 23rd Int. Conf. on Data Engineering*, pages 836–845, 2007.

[11] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. In *Networks*, 1972.

[12] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. 2008 ACM SIGMOD Int. Conf. On Management of Data*, pages 927–940, 2008.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 305–316, 2007.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style keyword search over relational databases. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 850–861, 2003.

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

[16] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 670–681, 2002.

[17] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 505–516, 2005.

[18] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 173–182, 2006.

[19] B. Kimelfeld and Y. Sagiv. New algorithms for computing steiner trees for a fixed number of terminals. In *http://www.cs.huji.ac.il/ bennyk/papers/steiner06.pdf*, 2006.

[20] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7), 1972.

[21] 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 *Proc. 2008 ACM SIGMOD Int. Conf. On Management of Data*, pages 903–914, 2008.

[22] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. 2006 ACM SIGMOD Int. Conf. On Management of Data*, pages 563–574, 2006.

[23] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 115–126, 2007.

[24] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 605–616, 2007.

[25] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: The power of rdbms. In *Proc. 2009 ACM SIGMOD Int. Conf. On Management of Data*, pages 681–694, 2009.

[26] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *Proc. 25th Int. Conf. on Data Engineering*, pages 724–735, 2009.

[27] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Scalable keyword search on large data streams. In *Proc. 25th Int. Conf. on Data Engineering*, pages 1199–1202, 2009.

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