Reachability Indexes for Relational Keyword Search

Alexander Markowetz, Yin Yang, Dimitris Papadias

Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{alexmar, yini, dimitris}@cse.ust.hk

***Abstract*— Due to its considerable ease of use, *relational keyword search* (R-KWS) has become increasingly popular. Its simplicity, however, comes at the cost of intensive query processing. Specifically, R-KWS explores a vast search space, comprised of all possible combinations of keyword occurrences in any attribute of every table. Existing systems follow two general methodologies for query processing: (i) *graph based*, which traverses a materialized data graph, and (ii) *operator based*, which executes relational operator trees on an underlying DBMS. In both cases, computations are largely wasted on graph traversals or operator tree executions that fail to return results. Motivated by this observation, we introduce a comprehensive framework for *reachability indexing* that eliminates such fruitless operations. We describe a range of indexes that capture various types of join reachability. Extensive experiments demonstrate that the proposed techniques significantly improve performance, often by several orders of magnitude.**

I. Introduction

Recently, keyword search has been extended to relational data (R-KWS) with considerable success. The main advantage of R-KWS lies in its ease of use, because it does not require any SQL skills or knowledge of the database schema. However, at the same time, R-KWS involves expensive query processing because all possible combinations of keyword occurrences in any attribute of every table must be explored. There are two query processing frameworks: *graph based* (GB) and *operator based* (OB). The former retrieves results by traversing an in-memory *data graph*. The latter executes a set of *relational operator trees*, each producing results of a certain structure. These trees are enumerated extensively, and collectively answer the query. Both approaches waste substantial resources on operations that do not contribute to the query. We observe that the chances to retrieve results are closely related to the connectivity along join sequences, and introduce a framework for *indexing reachability*. The proposed data structures eliminate useless graph traversals / operator trees prior to their execution, and achieve significant savings, commonly ranging in the order of magnitudes.

II. Preliminaries and Related Work

We assume a set of relational tables $R = \{R_1 \dots R_{|R|}\}$, accompanied by a schema graph S. Each node in S corresponds to a table R_i; two nodes R_i and R_j are connected through an edge, iff a common join condition has been defined between the two tables. We consider that edges in S are undirected. We denote the a^{th} tuple in table R_i as $r_{i,a}$. N signifies the set of all tuples: $N = R_1 \cup \dots \cup R_{|R|}$. Records may contain textual attributes. The dictionary of terms appearing anywhere in the database is denoted as $T = \{t_1 \dots t_{|T|}\}$. R-KWS semantics are based on a graph model of the database. Each node $r_{i,a}$ in this data graph G represents a record of R_i. There is an edge between two nodes $r_{i,a}$ and $r_{j,b}$, iff (i) R_i and R_j are directly connected in S, and (ii) $r_{i,a}$, $r_{j,b}$ can be joined according to the corresponding condition. Similar to the schema graph S, G is undirected. Figure 1 depicts a data graph for eleven records, according to the schema on its left. Tuple $r_{1,2}$ joins with $r_{2,1}$ and $r_{2,2}$, whereas $r_{1,1}$ does not have a join partner. Seven terms $T = \{t_1 \dots t_7\}$ are denoted next to the tuples in which they occur. The output of an R-KWS query $\{k_1 \dots k_m\}$ is commonly defined as the set of connected acyclic sub-graphs in G that (i) include all keywords, and (ii) are minimal, i.e., do not contain unnecessary trailing nodes. For instance, $(r_{4,4}, r_{2,2}, r_{3,3})$ is a result for $q = \{t_1, t_3, t_5\}$.

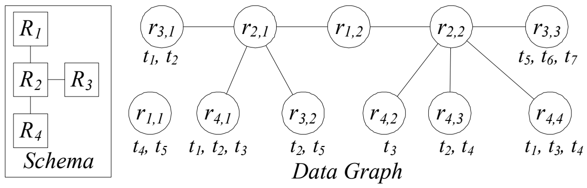


Fig. 1 Schema graph S and data graph G

GB systems process keyword queries through (i) an in-memory data graph G, and (ii) a disk resident inverted index (IID), mapping terms to tuples. Without loss of generality, we assume a breath-first graph traversal, initiated at each tuple r containing keyword k_1 (such tuples are retrieved through the IID). If a traversal encounters all remaining terms $k_2 \dots k_m$, GB constructs a result, consisting of r and nodes along paths to the other keywords. In contrast, OB systems generate a set of join expressions, collectively answering the R-KWS query. Each expression translates into a relational operator tree that produces results of a particular type. We use $R_i[T_x]$ to denote the set of records $r_{i,a} \in R_i$ that contain all terms $t \in T_x$, e.g., $R_4[t_1, t_3]$ signifies tuples of R_4 with t_1 and t_3 (i.e., $\sigma\{t_1,t_3\}R_4$). The join tree $R_4[t_1, t_3] \bowtie R_2 \bowtie R_3[t_5]$ retrieves the result $(r_{4,4}, r_{2,2}, r_{3,3})$.

R-KWS was introduced with *Banks* ([3], [14]), a GB system. Kimelfeld et al. [8], [16] propose a graph traversal that produces results within polynomial delay. *Blinks* [13] accelerates graph traversal through an index that maps nodes to reachable terms, and vice versa. It also partitions G and introduces a second level index. OB systems include *DBXplorer* [1], *Discover* ([10], [11]), *Kite* [21], and *Spark* [18]. Several of these systems impose a threshold T_{max} in order to eliminate long join sequences. Given a set of distributed databases, [23] and [26] use summaries to determine the most promising source to direct an R-KWS query. *Ease* [17] answers R-KWS queries by building an inverted index over all sub-graphs in G. Several authors propose ranking schemes, favoring short join sequences ([3], [11]) and adapting IR-based measures ([4], [5], [8], [10], [17], [18], [19]). Ding et al. [7] use dynamic programming to retrieve top-k results. Furthermore, R-KWS has been extended to XML (e.g., [9], [12], [25]), and relational data streams [20].

There is significant work on indexing reachability for large graphs. *Labeling schemes* associate nodes with labels, so that connectivity can be inferred by inspecting these labels. For undirected graphs, a label $r_{i,a}.L$ stores the ID of $r_{i,a}$'s connected component. For cyclic directed graphs, Cohen et al. [6] propose *dual labeling*, where each node $r_{i,a}$ receives two label sets $r_{i,a}.L_{in}$ and $r_{i,a}.L_{out}$, consisting of (a selection) of nodes connected to $r_{i,a}$ via inbound or outbound paths. An assignment is a *2-hop cover*, iff for every node $r_{i,a}$ that reaches some $r_{j,b}$, we have $r_{i,a}.L_{out} \cap r_{j,b}.L_{in} \neq \varnothing$. Unfortunately, finding a minimal 2-hop cover is NP hard, leading to several approximation schemes and heuristics (e.g., [6], [22], [24]).

III. Reachability Indexing

Reachability indexing aims at eliminating unproductive graph traversals and fruitless operator trees. Sections III-A and III-B focus on indexes for GB and OB, respectively. Section III-C discusses index compression. For our discussion, we assume that each result has a maximum size of T_{max}.

A. GB Indexing

Given a node $r_{i,a}$, the goal of GB indexing is to predict whether $r_{i,a}$ participates in any result of q. If not, all graph traversals involving $r_{i,a}$ can be omitted. *False dismissals* are not permitted; i.e., graph traversals can only be eliminated, if they are guaranteed not to generate output. On the other hand, *false positives* (unproductive graph traversals that are not eliminated) are permissible, although not desirable. We write $r_{i,a} \leftrightarrow r_{j,b}$ to signify that tuples $r_{i,a}$ and $r_{j,b}$ are connected through a path of at most T_{max} nodes. A *node-to-term* (N2T) index maps tuples to reachable terms. Formally, an N2T index is a function:

$N2T: N \to 2^T$

$N2T(r_{i,a}) = \{t \in T \mid \exists\, r_{j,b} \in \text{ in any } R_j[t], \text{ and } r_{i,a} \leftrightarrow r_{j,b}\ \}$.

It can be materialized as either (i) a set of reachability lists, or (ii) a reachability matrix. In the first approach, the index is a set of $|N|$ lists, one for every node $r_{i,a}$. The list of $r_{i,a}$ contains the terms that can be reached from $r_{i,a}$. The reachability matrix contains a bitmap of length $|T|$ for each node $r_{i,a}$. Figure 2 illustrates the lists and (partial) matrix for $r_{1,2}$ and $r_{2,1}$, assuming the data of Figure 1 and $T_{max} = 3$. The use of an N2T index during GB is straightforward. Assume node $r_{i,a}$ (retrieved through an inverted index) chosen for initiation of a graph traversal. If $r_{i,a}$ cannot reach some keyword in $\{q \setminus r_{i,a}.terms\ \}$, it is guaranteed not to participate in any result. Therefore, the traversal is avoided. As we show in Section III-C, all the proposed indexes (including N2T) can be effectively compressed using bloom filters.

$[r_{1,2}]$: $t_1, t_2, t_3, t_4, t_5, t_6, t_7$
$[r_{2,1}]$: t_1, t_2, t_3, t_5

(a) Lists for $r_{1,2}$ and $r_{2,1}$

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$r_{1,2}$	1	1	1	1	1	1	1
$r_{2,1}$	1	1	1	0	1	0	0

(ii) Bitmaps for $r_{1,2}$ and $r_{2,1}$

Fig. 2 N2T index representations

B. OB Indexing

Given an operator tree for q, the goal of OB indexing is to predict whether it returns any result. If not, the operator tree is discarded without execution. N2T is not effective for OB because it only requires tuples to reach a term t in *any* table. In contrast, the following *node-to-$R_i[t]$* (N2R) index restricts terms to *a certain table*. Formally, an N2R index constitutes a function:

$N2R: N \times R \to 2^T$

$N2R(r_{i,a}, R_j) = \{t \in T \mid \exists\, r_{j,b} \in R_j[t], \text{ and } r_{i,a} \leftrightarrow r_{j,b}\ \}$.

N2R consists of $|R|$ different N2T indexes, each storing terms reachable in tuples of a table R_j. Figure 3 illustrates the detection of unproductive operator trees through an N2R index. Let *op-tree* be an operator tree containing $R_i[T_x]$ and $R_j[T_y]$. If *op-tree* returns a result, each selection $R_i[T_x]$ includes a tuple that reaches all terms T_y in R_j (for every combination of $R_i[T_x]$ and $R_j[T_y]$). Conversely, if (for some combination) there is no record in $R_i[T_x]$ that reaches all terms T_y in R_j, *op-tree* is unproductive and eliminated without processing.

```
N2R _check(operator tree op-tree)
1. For every combination of selection operators $R_i[T_x]$ and $R_j[T_y]$
2.    Use inverted index to retrieve IDs of tuples in $R_i[T_x]$
3.    Boolean tmp = false
4.    For each $r_{i,a} \in R_i[T_x]$
5.       If $T_y \subset N2R(r_{i,a},R_j)$ //a tuple in $R_i[T_x]$ can reach all terms of $T_y$ in $R_j$
6.          tmp = true
7.    If (tmp = false) // $R_i[T_x]$ and $R_j[T_y]$ cannot reach each other
8.       Return "unproductive"
9. Return "requires execution" // all other cases
```

Fig. 3 OB-check using an N2R index

Consider a query involving terms $\{t_1, t_2, t_3\}$ and an operator tree with two selections $R_i[t_1]$ and $R_j[t_2, t_3]$, where no tuple of the former reach one in the latter. Yet, assume that $r_{i,a} \in R_i[t_1]$ reaches records $r_{j,b} \in R_j[t_2]$ and $r_{j,c} \in R_j[t_3]$. The call $N2R(r_{i,a}, R_j)$ returns both t_2 and t_3, but the tree is fruitless because the terms exist in different tuples. This case corresponds to a false positive (i.e., the tree is executed despite being unproductive). The following *node-to-node reachability index* (N2N) eliminates such false positives. For every record $r_{i,a}$, N2N stores the set of nodes in each table, which can be reached from $r_{i,a}$ in less than T_{max} hops. Formally:

$N2N: N \times R \to 2^N$

$N2N(r_{i,a}, R_j) = \{r_{j,b} \in R_j \mid r_{i,a} \leftrightarrow r_{j,b}\}$.

Assume, $T_{max} = 3$ and the data graph of Figure 1. Probed with $(r_{2,1}, R_3)$, the N2N index returns $r_{3,1}$ and $r_{3,2}$. Reachability checking is similar to that for N2R (in Figure 3). The main difference is in Line 5, where the condition becomes: $N2N(r_{i,a}, R_j) \cap R_j[T_y] \neq \varnothing$, i.e., the tree may yield results, if there is $r_{i,a} \in R_i[T_x]$ that can reach a record of R_j containing all terms in T_y (for every combination of $R_i[T_x]$ and $R_j[T_y]$). Compared to

N2R, the N2N is more effective (i.e., it eliminates more false positives), but it is larger because usually the number of reachable tuples is higher than that of reachable terms.

Both N2R and N2N require inverted indexes to identify records with keywords. Long inverted lists in very large databases may render index access expensive. The following R2R index works *without inverted indexes*. It indicates if any record in a given $R_i[t_1]$ reaches a tuple of a certain $R_j[t_2]$. An R2R index constitutes a function:

$R2R$: $R \times T \times R \to 2^T$

$R2R(R_i, t, R_j) = \{t' \in T \mid \exists r_{i,a} \in R_i[t], r_{j,b} \in R_j[t'], \text{ and } r_{i,a} \leftrightarrow r_{j,b}\}$.

Assume an operator tree returning a result. Every selection $R_i[T_x]$ produces a record that can reach a tuple from every other selection $R_j[T_y]$. Thus, for every term $t_1 \in T_x$ the call to $R2R(R_i, t_1, R_j)$ returns all terms $t_2 \in T_y$. Conversely, if the index fails to return all terms in T_y, the operator tree is guaranteed to remain unproductive and can safely be dropped.

C. Index Compression

We compress indexes into *Bloom filters* (BF) [2]. A BF is a summary for a *set* of objects. Instead of storing actual objects, it maintains an array of b bits, to which a set of hash functions $f = (f_1, ..., f_{|f|})$ maps objects. An object o is hashed with each function $f_i \in f$, and the bit at position $f_i(o)$ is set to non-zero. When verifying whether object o' is contained in *set*, o' is similarly hashed. If any bit at position $f_i(o')$ is zero, o' is certainly not part of *set*. If all bits are non-zero, we assume that o' is present. However, all $|f|$ bits may have been set by other objects, in which case o' corresponds to a *false positive*.

We apply BF as follows. A *reachability fact* signifies that two objects reach each other; e.g. the reachability records $r_{i,a}$ and $r_{j,b}$ can be expressed by the quadruple (i, a, j, b). The above indexes thus constitute collections of such facts. For instance, for an R2R index, we insert an object (i, x, j, y) into the BF, whenever we find tuples of $R_i[t_x]$ and $R_j[t_y]$ within distance $< T_{max}$; i.e., we hash (i, x, j, y), and set the corresponding bits. To check if tuples of $R_i[t_x]$ and $R_j[t_y]$ reach each other, we probe for (i, x, j, y) with the same hash functions. BF-compressed indexes compactly fit in memory, and a few Mbytes suffice even for large datasets.

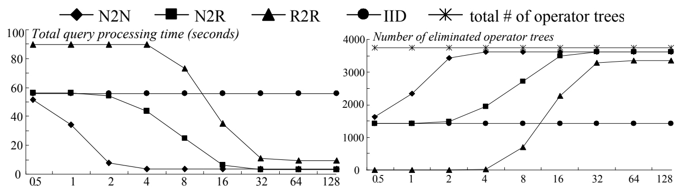
IV. Experiments

We experiment on a 2.2 GHz Core 2 Duo processor with MySQL 5.0. Indexes are implemented using C++. We assume a schema of eight tables in the shape of a ternary tree. Each table holds 10,000 tuples, containing four numeric attributes, on which we join with neighboring relations. Values range uniformly in [1, dr]; two tuples thus join with probability $js = 1/dr$ (dr signifies the domain range). A record contains TS random terms from a dictionary of $|T|$ = 100,000 entries. We assume term frequencies to follow Zipf's law, and assign term-IDs correspondingly; e.g., t_n appears with $1/n^{th}$ of the frequency of t_1. A query contains m keywords, chosen from $\{t_{150}, t_{151}, \ldots, t_{180}\}$; terms of higher frequency (e.g., t_3) are usually stop-words, while infrequent terms are rarely searched for. Indexes are stored as in-memory Bloom filters of b Mbytes. Table I lists the parameters under investigation.

TABLE I
Parameters Under Investigation

Parameter	**Range & Default**
b: size of BF	0.5, 1, 2, 4, 8, **16**, 32, 64, 128 (Mbytes)
TS: # terms per tuple	6, 8, **10**, 12, 14
T_{max}: result size threshold	2, 3, **4**, 5, 6
m: # query keywords	2, **3**, 4, 5
js: join selectivity (=1/dr)	1/5000, 1/7500, **1/10000**, 1/15000, 1/20000

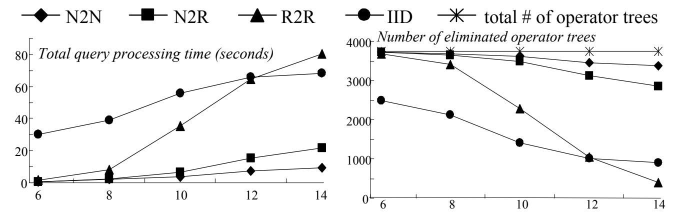
First, we evaluate N2N, N2R and R2R for OB query processing, and compare them against a baseline method (IID) that uses an inverted index to detect and eliminate operator trees with an empty input. We report (a) the processing time, including filtering and executing operator trees, and (b) the number of eliminated trees. Figure 4 shows both parameters as a function of the size b of the BF. Query processing is inversely proportional to the eliminated operator trees. Index effectiveness increases with b, until it stabilizes, when the index acts as if it were uncompressed. N2N reaches this point at 2 Mbytes, followed by N2R (16 Mbytes), and R2R (32 Mbytes). All indexes are significantly more effective than IID. N2N eliminates the majority of unproductive operator trees, and consistently outperforms IID even for a small BF.



(a) query processing time (b) eliminated operator trees

Fig. 4 Effect of Bloom filter size b

Figure 5 addresses the tuple size TS. The effectiveness of all methods drops, as TS increases, due to a raised chance for a tuple to contain a query keyword. R2R is particularly sensitive, as the number of reachable objects increases fast. Since the BF has a fixed length, this leads to a large number of false positives, resulting in a drop of effectiveness for TS=14. Other indexes behave better. N2N is the most effective, removing nearly all unproductive operator trees. With one exception (R2R at TS = 14), every proposed index outperforms IID.



(a) query processing time (b) eliminated operator trees

Fig. 5 Effect of the tuple size TS

Figure 6 evaluates the maximal number of tuples per result. A higher value of T_{max} increases the number of reachable tuples. When T_{max} exceeds 5, most pairs of tuple sets $R_i[t_x]$ and $R_j[t_y]$ are connected, and the effectiveness of R2R drops sharply. Generally speaking, the number of operator trees, and consequently the processing time (and potential savings) increases with T_{max}.

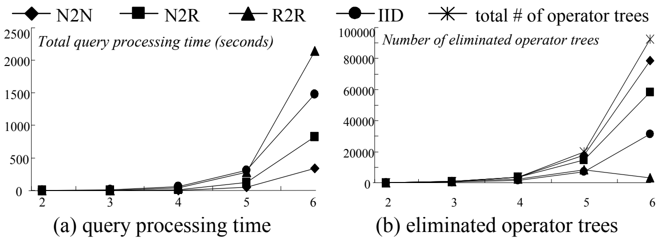


Fig. 6 Effect of T_{max}

Figure 7 illustrates the effect of the query length *m*. An increase of this parameter leads to an exponential growth in the number of operator trees. At the same time, unproductive operator trees become easier to detect. The benefits of reachability indexing hence increase sharply with *m*. Again, N2N excels in terms of effectiveness and efficiency.

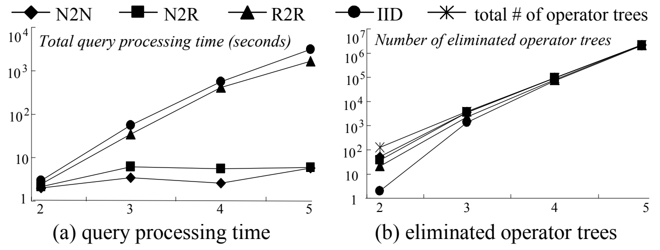


Fig. 7 Effect of the query length *m*

As shown in Figure 8, all indexes become more effective for high selectivity *js*. The number of unproductive operator trees is proportional to the domain range (*dr*), as it is increasingly difficult to form a connection between two tuples. For *dr*=15,000 our indexes eliminate most unproductive trees.

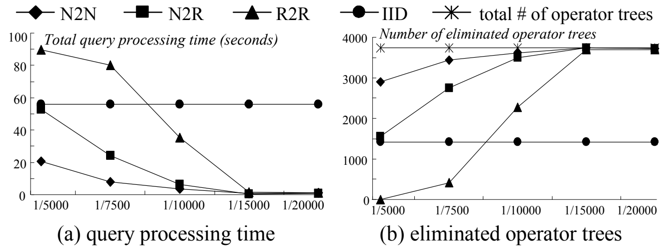


Fig. 8 Effect of join selectivity *js*

Finally, Figure 9 studies the impact of N2T on the number of graph traversals required by plain GB compared to GB+N2T, for T_{max} and *m*. Neither parameter influences the number of traversals prior to filtering with the N2T index. Yet, an increase (decrease) of *m* (T_{max}) makes tuples less likely to reach terms, and N2T's advantage becomes more pronounced.

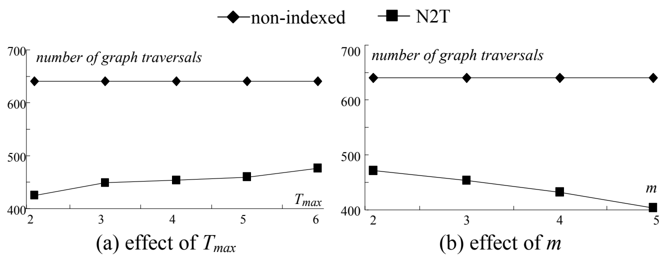


Fig. 9 Impact of the N2T index on GB

V. Conclusion

Current R-KWS methodologies waste CPU for fruitless graph traversals (GB) and unproductive operator trees (OB). We eliminate such operations though indexes that capture join reachability, specifically N2N, N2R and R2R for OB and N2T for GB systems. All structures readily fit into existing architectures and can be compacted into memory, without compromising accuracy. Extensive experiments demonstrate significant performance gains over conventional approaches.

References

[1] Agrawal, S., Chaudhuri, S., Das, G. "DBXplorer: A system for keyword-based search over relational databases", *ICDE*, 2002.

[2] Bloom, B. H., "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, 13(7): 422–426, 1970.

[3] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrbarti, S., Sudarshan, S. "Keyword searching and browsing in databases using BANKS", *ICDE*, 2002.

[4] Balmin, A., Hristidis, V., Papakonstantinou, Y. "ObjectRank: authority-based keyword search in databases", *VLDB*, 2004.

[5] Chaudhuri, S., Das, G., Hristidis, V., Weikum, G. "Probabilistic ranking of database query results", *VLDB*, 2004.

[6] Cohen, E., Halperin, E., Kaplan, H., Zwick, U. "Reachability and distance queries via 2-hop labels source", *SIAM Journal on Computing*, 32(5): 1338–1355, 2003.

[7] Ding, B., Yu, J. X., Wang, S., Qin, L., Zhang, X., Lin, X. "Finding top-k min-cost connected trees in databases", *ICDE*, 2007.

[8] Golenberg, K., Kimelfeld, B., Sagiv, Y. "Keyword proximity search in complex data graphs", *SIGMOD*, 2008.

[9] Guo, L., Shao, F., Botev C., Shanmugasundaram J. "XRANK: ranked keyword search over XML documents", *SIGMOD*, 2003.

[10] Hristidis, V., Gravano, L., Papakonstantinou, Y. "Efficient IR-style keyword search over relational databases", *VLDB*, 2003.

[11] Hristidis, V., Papakonstantinou, Y. "DISCOVER: keyword search in relational databases", *VLDB*, 2002.

[12] Hristidis, V., Papakonstantinou, Y., Balmin, A. "Keyword proximity search on XML graphs", *ICDE*, 2003.

[13] He, H., Wang, H., Yang, J., Yu, P., "BLINKS: ranked keyword searches on graphs", *SIGMOD*, 2007.

[14] Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H. "Bidirectional expansion for keyword search on graph databases", *VLDB*, 2005.

[15] Katz, M., Katz, N., Korman, A., Peleg, D. "Labeling schemes for flow and connectivity", *SODA*, 2002.

[16] Kimelfeld, B., Sagiv, Y., "Finding and approximating top-k answers in keyword proximity search", *PODS*, 2006.

[17] Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L. "EASE: efficient and adaptive keyword search on unstructured, semi-structured and structured data", *SIGMOD*, 2008.

[18] Luo, Y., Lin, X., Wang, W., Zhou, X. "SPARK: Top-*k* keyword query in relational databases", *SIGMOD*, 2007.

[19] Liu, F., Yu, C., Meng, W., Chowdhury, A. "Effective keyword search in relational databases", *SIGMOD*, 2006.

[20] Markowetz, A., Yang, Y., Papadias, D., "Keyword search on relational data streams", *SIGMOD*, 2007.

[21] Sayyadian, M., LeKhac, H., Doan, A., Gravano, L. "Efficient keyword search across heterogeneous relational databases", *ICDE*, 2007.

[22] Schenkel, R., Theobald, A., Weikum, G. "Efficient creation and incremental maintenance of the HOPI index for complex XML document collections", *ICDE*, 2005.

[23] Vu, Q. H., Ooi, B. C, Papadias, D., Tung, A., "A graph method for keyword-based selection of the top-k databases", *SIGMOD*, 2008.

[24] Wang, H., He H., Yang, J., Yu, P., Yu, J. X. "Dual labeling: answering graph reachability queries in constant time", *ICDE*, 2006.

[25] Xu, Y., Papakonstantinou, Y. "Efficient keyword search for smallest LCAs in XML databases", *SIGMOD*, 2005.

[26] Yu, B., Li., G., Sollins, K., Tung, A. "Effective keyword-based selection of relational databases", *SIGMOD*, 2007.