A Graph Method for Keyword-based Selection of the top-K Databases

Quang Hieu Vu [1], Beng Chin Ooi [1], Dimitris Papadias [2], Anthony K. H. Tung [1]
[1] National University of Singapore *, [2] Hong Kong University of Science and Technology †
hieuvq@nus.edu.sg, ooibc@comp.nus.edu.sg, dimitris@cs.ust.hk, atung@comp.nus.edu.sg

ABSTRACT

While database management systems offer a comprehensive solution to data storage, they require deep knowledge of the schema, as well as the data manipulation language, in order to perform effective retrieval. Since these requirements pose a problem to lay or occasional users, several methods incorporate *keyword search* (KS) into relational databases. However, most of the existing techniques focus on querying a single DBMS. On the other hand, the proliferation of distributed databases in several conventional and emerging applications necessitates the support for keyword-based data sharing and querying over multiple DMBSs. In order to avoid the high cost of searching in numerous, potentially irrelevant, databases in such systems, we propose *G-KS*, a novel method for selecting the top-K candidates based on their potential to contain results for a given query. *G-KS* summarizes each database by a *keyword relationship graph*, where nodes represent terms and edges describe relationships between them. Keyword relationship graphs are utilized for computing the similarity between each database and a KS query, so that, during query processing, only the most promising databases are searched. An extensive experimental evaluation demonstrates that *G-KS* outperforms the current state-of-the-art technique on all aspects, including precision, recall, efficiency, space overhead and flexibility of accommodating different semantics.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Distributed Databases*

General Terms

Algorithms, Management, Performance

Keywords

Distributed Databases, Relational Databases, Keyword Search, Database Summary, Graph, Information Retrieval

*Supported in part by ASTAR SERC Grant 072 101 0017 as part of S3 project [22].

†Supported by grant HKUST 6184/06E from Hong Kong RGC.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

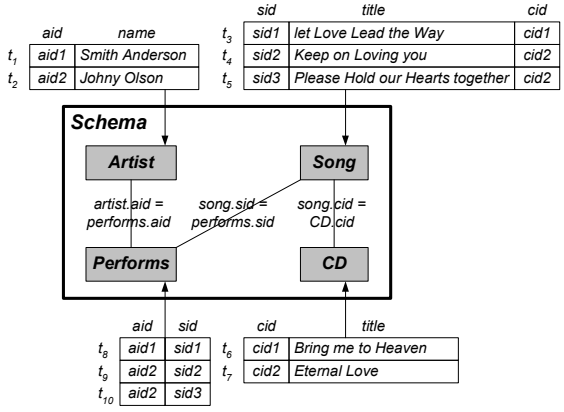


Figure 1: Running example of a music database

1. INTRODUCTION

Users of conventional DBMSs should have good knowledge of both the database schema and the data manipulation language (e.g., SQL) in order to effectively issue queries. Since the schema and the query language may be complex, database systems are often limited to advanced users. Several methods aim at decreasing this complexity by providing keyword-search (KS) functionality over a single database [3, 1, 14, 13, 15, 17, 12, 19]. In relational KS the basic unit of information is a tuple and each result is a set of tuples that (i) contain all (or most) keywords, and (ii) can be joined together in a meaningful way (usually on *Primary Key - Foreign Key* relationships). This contrasts traditional information retrieval (IR), where each result is a single document (i.e., a basic unit of information).

We illustrate relational KS using the music database DB_1 of Figure 1, where edges between the tables correspond to join conditions. Given the KS query $q = \{Anderson, love\}$, the system should return a result $t_1 \bowtie t_8 \bowtie t_3$, signifying that there is an artist called $Anderson$ (t_1) who performs (t_8) a song (t_3) containing $love$ in its title. Note that although the term $love$ also appears in t_4 and t_7, these tuples do not generate results as they cannot be connected to a record (i.e., t_1) containing $Anderson$. In the general case, there may exist numerous results, which can be ranked according to various criteria. A common scoring criterion used in all existing techniques is the *distance* [1], i.e., the number of joins

[1]Note that the distance is not constrained by the schema, but depends on the specific table instances. For example, in the schema of Figure 1 tuples could be connected by chains of

between two tuples containing the query keywords. In DB_1, the distance between t_1 and t_3 is 2. Intuitively, the smaller the distance the more related the tuples, and therefore, the higher the importance of the corresponding result.

Relational KS liberates the user from having to learn SQL and study the database schema (i.e., the tables, the attributes, and their possible connections). On the other hand, as keywords may appear in arbitrary columns (e.g., the term *love* may exist in the title of a song or an album, or even in the name of an artist), query processing is significantly more expensive than conventional DBMSs because all possible combinations of keyword co-occurrences have to be explored. The advantages (i.e., flexibility) and shortcomings (i.e., efficiency) of relational KS are amplified in distributed systems integrating multiple databases DB_1, ..,DB_D. For instance, in the context of our P2P system BestPeer [2], a group of peer nodes belonging to different universities can cooperate. If they wish to share their library databases, keyword based data sharing and search render the different schemas transparent, so that users can issue KS queries using a common interface without any knowledge of the underlying structure of DB_1, ..,DB_D. However, processing each query in all the individual databases, and then combining the partial results, could be very costly and potentially unnecessary, especially if the final answers exist in a small subset of the databases.

In traditional IR, distributed systems alleviate the efficiency issue by pre-computing a term summary for each document repository. Given a query q, the system can select the most appropriate collections for processing q based on the similarity between q and the summaries. In other words, summaries act as documents in the corpus formed by all repositories and provide a fast mechanism for filtering out non-promising sources. However, IR methods are insufficient for relational KS as the mere presence of the query keywords in a summary is not indicative of the existence of results in the corresponding database; in addition, the tuples containing these keywords should be *connected* in a meaningful way. Assume for instance, a system integrating DB_1 of Figure 1 with another database, say DB_2, that includes the same tables and tuples as DB_1 except for t_8. Given $q = \{Anderson, love\}$, both summaries of DB_1 and DB_2 contain similar statistical information (e.g., frequency, inverse term frequency) for the keywords *Anderson* and *love*, implying that they would be equally good candidates for search. However, the absence of t_8 in DB_2 eliminates the result $t_1 \bowtie t_8 \bowtie t_3$. In general, IR-based methods do not capture the inherent structure of a DBMS because they ignore connectivity and distance information among tuples containing terms.

In this paper, we aim at selecting the top-K databases for processing a KS query, where K is an input parameter. In accordance with the methodology of distributed IR techniques, we assume a system that pre-processes and maintains summaries of several DBMSs DB_1, DB_2,..,DB_D. Given a KS query q, the system directs q only to the $K < D$ databases most likely to contribute results, in order to minimize to total processing cost of q without sacrificing precision and recall. The only existing technique focusing on the same problem is *M-KS* [27], which is based on the concept of the *keyword relationship matrix* (KRM). Specifically, at a pre-processing phase, *M-KS* builds a KRM_l for every DB_l, which acts as its summary. *For each term pair* (k_i,k_j) there is an entry $KRM_l(k_i,k_j)$ that records the frequencies of occurrences of the two terms at different *distances*. Given the 14 terms in DB_1 [2], KRM_1 would contain 14x14 entries. The entry $KRM_1(Olson,love)$ stores that the two terms can be connected once at distance 2 (i.e., $t_2 \bowtie t_9 \bowtie t_4$) and once at distance 3 ($t_2 \bowtie t_{10} \bowtie t_5 \bowtie t_7$). The similarity between a query q and DB_l is computed using the KRM_l entries of all possible keyword pairs in q; e.g., if q=$\{k_1, k_2, k_3\}$, the score of DB_l is based on $KRM_l(k_1,k_2)$, $KRM_l(k_1,k_3)$ and $KRM_l(k_2,k_3)$.

Despite its significant performance gains with respect to the naive solution of processing a KS query in all databases, *M-KS* has several disadvantages. First, it uses only binary relationships between keyword terms to eliminate non-promising databases. Consequently, it yields numerous *false positives* for queries where all pairs of keywords are related, but there is no join sequence linking the binary connections in a single result. Second, since KRM records only the frequency of term co-occurrences (but no additional statistics), *M-KS* is unsuitable for ranking based on IR measures (e.g., those in [17]). Finally, *M-KS* does not include a mechanism for handling OR semantics. Therefore, it cannot retrieve (potentially interesting) answers even when a single keyword is missing.

To overcome these shortcomings, we propose *G-KS*, a graph based method for supporting keyword based data sharing and search over multiple databases. *G-KS* summarizes each database DB_l as a *keyword relationship graph* KRG_l that captures the terms and their relationships with weighted nodes and edges. KRG_l minimizes the chance of false positives by imposing more stringent conditions than simple binary relationships. Based on KRG_l, *G-KS* can effectively estimate the importance of DB_l with respect to a query under both the AND and the OR semantics. Our experiments show that, compared to *M-KS*, *G-KS* (i) is more effective (improving recall/precision by as much as 50%), (ii) is more efficient (reducing query processing cost by as much as 50%), (iii) incurs less pre-processing time (faster by 30%), and (iv) has less space overhead (smaller by 17%). Our contributions are summarized as follows:

- We propose *G-KS*, a novel graph-based method for relational KS over multiple databases. We show how to construct, maintain and compress the *KRGs* in order to reduce the pre-processing and storage overhead.
- We present an IR-inspired method to compute the importance of nodes and edges, and an algorithm to estimate the potential of a join solution containing all query keywords.
- We confirm the efficiency and effectiveness of *G-KS* through extensive experiments with real datasets in a client-server architecture.

The rest of the paper is organized as follows. Section 2, discusses related work. Section 3, presents the keyword relationship graph model. Section 4 describes the construction

more than 3 joins through intermediate records that belong to the same table. Most relational KS systems (e.g., [14, 27]) use a distance threshold to eliminate long chains of joins, which usually lead to uninteresting results.

[2]Terms start with capital letters in Figure 1. Note that *let*, *the*, *on*, *you*, *our*, *together*, *me*, *to* are stop words, and hence they are not considered terms.

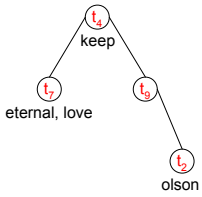


Figure 2: Connection tree for query {*Olson*, *keep*, *eternal*, *love*}

and maintenance of *KRGs*. Section 5 introduces the query processing algorithm and discusses the advantages of *G-KS* over *M-KS*. Section 6 compares the two methods experimentally and Section 7 concludes the paper.

2. RELATED WORK

Section 2.1 overviews related work on KS over a single system. Section 2.2 surveys KS on multiple data repositories.

2.1 Keyword search over a single system

The majority of IR literature has focused on KS for unstructured documents stored at a single repository. Most methods are based on the Vector Space Model [25], which represents documents as term vectors [23] in the Cartesian Space. Each element in a vector captures a term and its weight, defined as $w = tf \cdot idf$, where tf is the *term frequency* in the document and *idf* is the *inverse document frequency* reflecting the general importance of the term in the entire corpus. Specifically, *idf* decreases the weight of terms that occur in numerous documents and, therefore, have low discriminating value (we refer to such terms as *popular terms*). Queries are also represented as vectors of keywords. The similarity of a query and a document is measured as the Cosine of the angle between their vector representations.

Several systems apply KS on a single relational DBMS. BANKS [3] creates a *data graph*, where each node represents a tuple, and edges connect tuples that can be joined (e.g., according to *Primary Key - Foreign Key* relationships). A KS query is processed by a graph traversal that searches for *connection trees* containing the query keywords. A connection tree is a Steiner tree in which every leaf node corresponds to a record containing at least one query keyword. Internal nodes represent tuples that connect the leaf records and may include no query keywords. Figure 2 shows a connection tree for query $q = \{Olson, keep, eternal, love\}$ on the example database of Figure 1. Intuitively, the tree captures connectivity information for a result, in this case $t_7 \bowtie t_4 \bowtie t_9 \bowtie t_2$. BANKS employs a backward search strategy from leaf nodes containing the query keywords towards the root. On the other hand, [15, 12] apply bi-directional search, which improves efficiency. DBXplorer [1] and DISCOVER [14] use a higher level of representation – *candidate networks* created from the schema of the database by join operations. The systems use the candidate networks to generate operator trees for evaluating the query. Both DBXplorer and DISCOVER rank results based exclusively on the distance of the tuples containing the query keywords, whereas [13, 17, 19] utilize state-of-the-art IR measures to calculate scores.

Besides relational databases, KS has been applied in the context of XML databases [11, 6, 16, 26, 18], where the search space is represented by a tree structure. Query results are the subtrees (or a part of the subtrees) of the XML document rooted at the lowest common ancestors (LCAs) [11], the semantic interconnection LCAs [6], the meaningful LCAs [16], or the smallest LCAs [26] of leaf nodes containing the query keywords. [18] presents a method to automatically select meaningful nodes in these subtrees. Recently, KS has also been extended to relational data streams [20], which necessitate the incorporation of temporal semantics and provision for frequent updates.

2.2 Keyword search over distributed systems

IR methods for KS over multiple repositories of documents generate summaries at a pre-processing phase. The summaries are utilized during query processing to filter out collections that cannot lead to good results. The existing techniques differ mainly on the way that they construct summaries. In the simplest form, GlOSS [10] and CVV [28] use only term frequencies to form summary information of each collection. CORI [4] adds a factor called *inverse collection frequency*, which reflects the importance of a term in all the repositories. Some other works [21, 9, 5] also consider dependency relationships between terms (e.g., between occurrences of *computer* and *programming*). Although such relationships exist in natural languages [8], they are not applicable in relational KS. [24] proposes a method for relational KS, where the tuples containing the query keywords may belong to *different* databases, assuming that records of distinct DBMSs can be joined. On the other hand, *M-KS* [27] considers that the various databases are independent and the keywords must exist in the same DBMS to form a result. Since *M-KS* focuses on the same setting as *G-KS* and is our only competitor, in the sequel we provide a more detailed description.

M-KS summarizes every DB_l with a keyword relationship matrix KRM_l capturing the binary relationships between terms at different distances. Specifically, each entry $KRM_l(k_i, k_j)$ stores a vector $d_0d_1...$, where d_0 is the number of times that terms k_i and k_j occur in the same tuple, d_1 is the number of times that they occur in tuples with distance 1 and so on. In order to compute these vectors (for all term pairs), *M-KS* scans DB_l, parses each tuple, removes stop words, stems the terms and inserts them into a table T_0 associating each term with the tuples that contain it. The d_0 values are obtained using T_0. Then, it joins all tables on foreign key relationships to generate a table T_1 with tuples at distance 1; based on T_1 and T_0, *M-KS* derives d_1 for all term pairs. Similarly, T_2 is produced by self-joining T_1, and is used to obtain d_2. For $m > 2$, T_m is the result of $T_{m-1} \bowtie T_1$ after excluding tuples that exist in T_1,..,T_{m-1}. The similarity between a query q and DB_l is based on the KRM_l entries of all possible keyword pairs in q. Assuming that q=$\{k_1, k_2, k_3\}$, *M-KS* retrieves $KRM_l(k_1,k_2)$, $KRM_l(k_1,k_3)$, $KRM_l(k_2,k_3)$ and assigns binary weights that reflect both the frequency of the co-occurrences and their distances (co-occurrences at smaller distance have a larger weight). The score of DB_l is computed as a function (e.g., sum, product, minimum) of the binary weights.

3. KEYWORD RELATIONSHIP GRAPHS

G-KS summarizes the terms and their relationships in each DBMS using a keyword relationship graph (*KRG*). Figure 3 illustrates the *KRG* for the example database of

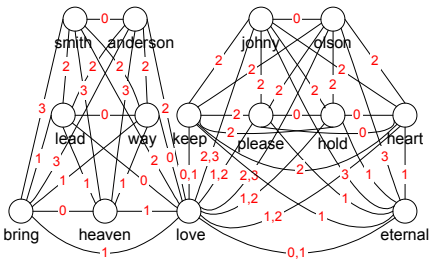


Figure 3: *KRG* for the example database

Figure 1. A node corresponds to a term and has a weight, which reflects its significance relative to other terms in the database. If two terms k_i and k_j exist in tuples t_x and t_y that can be connected through a sequence of joins, there is an edge between their corresponding nodes in the graph. The distance d between t_x and t_y (in terms of join [3] operations) is marked on the edge. When k_i and k_j can be connected through multiple paths of variable distances, each distinct value of d is recorded. For instance, the edge between *Olson* and *love* has two values: 2 due to the result $t_2 \bowtie t_9 \bowtie t_4$, and 3 due to $t_2 \bowtie t_{10} \bowtie t_5 \bowtie t_7$. Every distance value in the graph is associated with a weight [4] that measures the importance of the connection. As opposed to *M-KS* that considers only frequency information, *G-KS* utilizes IR-inspired measures to assign weights. Sections 3.1 and 3.2 discuss the computation of weights for nodes and edges, respectively. Section 3.3 presents a method for compressing the *KRG*. Table 1 summarizes the frequent symbols used throughout this paper for easy reference.

3.1 Weight of a node

Let *DB* be a database and k_i be a term appearing in a tuple t of *DB*. We use $c_i(t)$ to denote the number of occurrences of k_i in t. The *size* of tuple t is $S(t) = \sum_k c_k(t)$. N is the cardinality of tuples that include terms in *DB*, and N_i is the number of tuples containing k_i.

DEFINITION 1. *Term frequency, $tf_i(t)$*
The term frequency of k_i in tuple t is $tf_i(t) = \frac{c_i(t)}{S(t)}$. □

DEFINITION 2. *Inverse tuple frequency, iuf_i*
The inverse tuple frequency of term k_i is $iuf_i = \ln \frac{N+1}{N_i}$. □

Intuitively, $tf_i(t)$ and iuf_i are analogous to term frequency and inverse document frequency in the Vector Space Model [25]. For instance, the sample database of Figure 1 contains N=7 tuples with terms (t_1 to t_7). The term *Anderson* has a single appearance in t_1 (i.e., $N_{anderson}$=1 and $c_{Anderson}(t_1)$=1), which contains 2 terms in total (i.e., $S(t_1)$ = 2). Given the above, $tf_{anderson}(t_1) = \frac{1}{2}$ and $iuf_{Anderson}$ = $\ln(\frac{8}{1})$.

DEFINITION 3. *Term weight, $w_i(t)$, Node weight, w_i*
The weight of term k_i in a tuple t is $w_i(t) = tf_i(t) \cdot iuf_i$. The weight of the node representing k_i is $w_i = \frac{\sum_{t=1}^{N_i} w_i(t)}{N_i}$. □

Observe that $w_i(t)$ measures the importance of a term in a tuple, whereas w_i captures the average score for a particular term among all tuples that contain it. Continuing the example, $w_{Anderson}(t_1) = tf_{Anderson}(t_1) \cdot iuf_{Anderson} = \frac{1}{2} \cdot \ln(\frac{8}{1}) = 1.040$. Since there is only one occurrence of the term in the database, the node representing *Anderson* in the *KRG* of Figure 3 has a weight $w_{Anderson} = w_{Anderson}(t_1)$. Similarly, the term *love* appears in N_{love}=3 tuples t_3, t_4, t_7. The sizes of these tuples are $S(t_3)$=3, $S(t_4)$=2 and $S(t_7)$=2, respectively. Consequently, the weight of the term in the *KRG* is $w_{love} = average(\frac{1}{3} \cdot \ln(\frac{8}{3}) + \frac{1}{2} \cdot \ln(\frac{8}{3}) + \frac{1}{2} \cdot \ln(\frac{8}{3})) =$ 0.436. The intuition behind $w_{Anderson} > w_{love}$ is that since *Anderson* exists in a single tuple, it has a larger discriminating value (and weight) than *love*, which occurs in 3 records (i.e., it is a popular term).

Table 1: Table of symbols

Symbol	Definition
t	tuple in a relational database
k	term in a relational database
n	node in a *KRG*
d	distance relationship
tf	term frequency
iuf	inverse tuple frequency
pf	pairwise term frequency
iwf	inverse pairwise frequency
w	weight (of term, node, edge)

3.2 Weight of an edge

Let t_x and t_y be two tuples containing terms k_i and k_j, respectively. These tuples may be connected by several paths of equal distances that pass through different intermediate records: $nc(t_x, t_y, d)$ is the number of unique connections between t_x and t_y at distance d, and $c_{ij}(t_x, t_y, d) = nc(t_x, t_y, d) \cdot c_i(t_x) \cdot c_j(t_y)$ is the number of connections between k_i and k_j in t_x and t_y. The values of $c_{ij}(t_x, t_y, d)$ and $nc(t_x, t_y, d)$ differ only if there are multiple occurrences of k_i (k_j) in t_x (t_y). $S_{ij}(t_x, t_y, d) = nc(t_x, t_y, d) \cdot S(t_x) \cdot S(t_y)$ is the number of connections between all term pairs in t_x and t_y at distance d.

DEFINITION 4. *Pairwise frequency, $pf_{ij}(t_x, t_y, d)$*
The pairwise frequency of two terms k_i and k_j in tuples t_x and t_y at distance d is $pf_{ij}(t_x, t_y, d) = \frac{c_{ij}(t_x, t_y, d)}{S_{ij}(t_x, t_y, d)} = \frac{c_i(t_x) \cdot c_j(t_y)}{S(t_x) \cdot S(t_y)} = tf_i(t_x) \cdot tf_j(t_y)$. □

Consider the terms *Olson* appearing in t_2 and *love* appearing in t_4. Recall that the two terms can be combined by two join operations connecting: $t_2 \bowtie t_9 \bowtie t_4$. Given that $tf_{Olson}(t_2) = tf_{love}(t_4) = 0.5$, we obtain $pf_{Olson,love}(t_2, t_4, 2)$ = 0.25. Note that nc is canceled out in the computation of $pf_{ij}(t_x, t_y, d)$, and therefore, its existence is not necessary for the definition of *pairwise frequency*. We chose to include nc for consistency with the previous section. Specifically, the fraction $\frac{c_{ij}(t_x, t_y, d)}{S_{ij}(t_x, t_y, d)}$ corresponds to $\frac{c_i(t)}{S(t)}$ used in the definition of the *term frequency*. Next, let $N_{ij}(d)$ be the total number of cases where two tuples containing k_i and k_j can be connected at distance d. Similarly, $N(d)$ is the total number of cases where two tuples containing any terms can be connected at distance d.

DEFINITION 5. *Inverse pairwise frequency, $iwf_{ij}(d)$*
The inverse pairwise frequency of two terms k_i and k_j at distance d is $iwf_{ij}(d) = \ln \frac{N(d)+1}{N_{ij}(d)}$ □

DEFINITION 6. *Pairwise weight, $w_{ij}(t_x, t_y, d)$, Edge weight, $w_{ij}(d)$*

[3] Similar to [14, 27], we may use a threshold to avoid long chains of joins.

[4] For simplicity, we omit the node and edge weights in Figure 3.

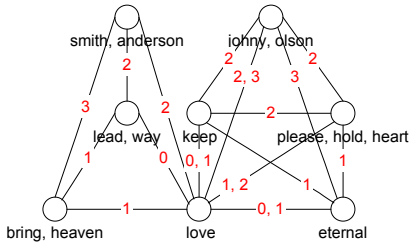


Figure 4: Compressed *KRG*

The pairwise weight of two terms k_i and k_j in tuples t_x and t_y at distance d is $w_{ij}(t_x, t_y, d) = pf_{ij}(t_x, t_y, d) \cdot iwf_{ij}(d)$. The weight of the edge connecting k_i and k_j at distance d is $w_{ij}(d) = \frac{\sum_1^{N_{ij}} w_{ij}(t_x,t_y,d)}{N_{ij}(d)}$. □

Intuitively, $w_{ij}(d)$ is the average weight among all pairs of tuples in the database that are connected at distance d and include k_i and k_j, respectively. Continuing our example, there is a single way to relate *Olson* and *love* at d=2; thus, $N_{olson,love}(2) = 1$. The total number of cases where tuples (containing terms) are connected by 2 join operations is $N(2) = 4$ ($t_1 \bowtie t_8 \bowtie t_3$, $t_2 \bowtie t_9 \bowtie t_4$, $t_2 \bowtie t_{10} \bowtie t_5$, and $t_4 \bowtie t_7 \bowtie t_5$). Given the above: $iwf_{Olson,love}(2) = \ln \frac{5}{1}$. Consequently, $w_{Olson,love}(t_2, t_4, 2) = 0.25 \cdot \ln \frac{5}{1} = 0.402$. Since there is no other result involving these terms at d=2, the corresponding weight for edge of the *KRG* in Figure 3 is $w_{Olson,love}(2) = w_{Olson,love}(t_2, t_4, 2)$.

3.3 Graph compression

Typically, a large percentage of the terms in a DBMS (around two thirds in our experiments) appear only once. If such terms occur in the same tuple: (i) they have the same weight; (ii) they have the same set of connections to other nodes; and (iii) these connections are of equal weight. Based on these observations we propose the following method for *KRG* compression. During pre-processing, *G-KS* identifies, for each tuple, the terms that appear only once in the database and inserts them in a compound node. Thus, the *KRG* has two types of nodes: *single* nodes containing one term and *compound* nodes consisting of multiple terms. The weight of a compound node, or an edge involving a compound node, is calculated using any of the included terms. This simple method decreases significantly the size of the graph without incurring any information loss. Figure 4 shows the compressed graph for the *KRG* of Figure 3. For instance, the terms *Johny* and *Olson*, which appear in DB_1 a single time in t_2, generate a compound node. Similarly, all three terms *please, hold, heart* of t_5 are unique and map to another compound node. In total, the reduced graph contains 8 nodes and 16 edges, compared to 14 nodes and 49 edges in the original *KRG*.

4. GRAPH CONSTRUCTION AND MAINTENANCE

Section 4.1 describes the initial construction of the keyword relationship graph. Section 4.2 discusses its maintenance in the presence of database updates.

4.1 Graph construction

The *KRG* of a database is constructed in three steps. Initially, *G-KS* extracts terms from the data tuples to create nodes for the *KRG*. In this step, the system creates a compound nodes for terms that occur only once in the database and in the same tuple. The remaining terms generate single nodes. At the end of this phase, weights are assigned based on the definitions of the previous section.

In the second step, the system constructs relationships between tuples in the database at different distances $d \geq 1$. We denote with $R_T(d)$ the set of tuple pairs (t_x, t_y) that can be connected at distance d. *G-KS* inserts a pair (t_x, t_y) in $R_T(1)$ if they have a *Primary Key - Foreign key* (PK-FK) relationship. For $d > 1$, (t_x, t_y) is added to $R_T(d)$, if there exists a tuple t_z such that (i) t_z has a relationship with either t_x or t_y in $R_T(d-1)$ and (ii) t_z can be joined with the other tuple of the pair. For instance, $(t_x, t_y) \in R_T(d)$, if $(t_x, t_z) \in R_T(d-1)$ and there is a PK-FK relationship between t_z and t_y.

Finally, based on the relationships of tuples at different distances, *G-KS* creates edges between nodes at the *KRG*. For each pair of nodes (n_i, n_j), where n_i represents a term in t_x, n_j represents a term in t_y, and $(t_x, t_y) \in R_T(d)$, there is an edge between n_i and n_j at distance d. Note that if the nodes correspond to terms in the same tuple, the distance is 0.

4.2 Graph maintenance

When new tuples are inserted or old tuples are deleted, the *KRG* should be modified accordingly to reflect the current instance of the database. Since an update can be considered as a deletion followed by an insertion, we only discuss the maintenance of *KRG* in case of insertions and deletions. When a tuple t is inserted into the database, *G-KS* extracts all the included terms. New terms lead to the creation of nodes in the *KRG*, while existing terms lead to recalculation of the weights of the corresponding nodes. Then, the system computes the distance relationship for all pairs (t, t_x) using PK-FK relationships. Additionally, a new entry for each pair (t_x, t_y) is created in $R_T(d_1 + d_2)$, if there exist $(t, t_x) \in R_T(d_1)$ and $(t, t_y) \in R_T(d_2)$.

Similar to the process of graph construction, based on the updated relationships of tuples at different distances, *G-KS* creates new relationships for the corresponding nodes. If these nodes were already connected at that distance, the corresponding weight is re-computed. Otherwise, a new (distance, weight) pair is added to the edge. If this is the first connection between the nodes, this is equivalent to adding an edge to the *KRG*. Finally, the system creates new relationships and updates the weight of the existing relationships at distance 0 for nodes representing terms in the inserted tuple.

When an existing record t is deleted, *G-KS* first extracts all terms in t and identifies the set I of nodes containing these terms. Let $I(d)$ be the set of nodes that have relationships with nodes in I at distance d. For each value of d, *G-KS* retrieves $I(d)$ and deletes relationships between nodes n_i and n_j at distance d satisfying one of the following conditions (i) $n_i, n_j \in I \wedge d = 0$ or (ii) $n_i \in I \wedge n_j \in I(d)$ or (iii) $n_i \in I(d_1) \wedge n_j \in I(d_2) \wedge d = d_1 + d_2$. For the third case, note that although the deletion of t may not affect (n_i, n_j) (if the two tuples have multiple connections at d), this cannot be known since the *KRG* does not maintain information about the specific paths. However, if n_i and n_j can be connected through a path not containing t, their relationship is recreated in the next step.

Next, all nodes that represent exclusively terms of t are removed from the KRG (i.e., the corresponding terms do not appear anywhere else in the databases) and the weights of the remaining nodes in I are updated. Finally, let T_I be the set of tuples containing the terms represented by nodes in I. Similarly, $T_I(d)$ is the set of tuples containing the terms corresponding to nodes in $I(d)$. *G-KS* computes new distance relationships between all tuple pairs (t_x, t_y) such that (i) $t_x \in T_I$ and $t_y \in T_I(d)$, or (ii) $t_x \in T_I(d_1)$ and $t_y \in T_I(d_2)$ and $d = (d_1 + d_2)$. Then, it updates edge weights based on the new tuple relationships. Finally, *G-KS* recreates relationships at distance 0 for nodes in I representing terms in the same tuple.

In order to avoid the cost of frequent revisions, the KRG may be updated in a batch mode periodically (e.g., at the end of each working day) or after the database modifications accumulate to a certain level. *G-KS* processes batch insertions using the same method as for a single record, except that the extracted terms exist in any of the new tuples. For batch deletions, the system applies the methodology of single deletion considering all terms of the deleted tuples.

5. QUERY PROCESSING

Given a KS query q and a set of $KRGs$, each created for a database, the goal of query processing is to find the top-K databases for directing the query. Section 5.1 introduces the concepts of join keyword trees and candidate graphs used to evaluate the potential of a database. Section 5.2 proposes an algorithm for finding candidate graphs. Section 5.3 describes the ranking mechanism, and Section 5.4 discusses the advantages of *G-KS* over *M-KS*.

5.1 Join keyword trees and candidate graphs

Let KRG be the keyword relationship graph of a database DB, and SG be a sub graph of KRG. We evaluate the potential of DB with respect to query q based on the structure of the KRG. Towards this direction we define the concept of the *join keyword tree* (JKT). To avoid confusion, we refer to a JKT node as a *vertex*.

DEFINITION 7. *Join keyword tree, $JKT(SG)$*
Given SG, *$JKT(SG)$ is a tree satisfying the following properties:*

1. *Each tree vertex tn_i maps to a non-empty set of nodes of* SG, *and the tree vertices should collectively contain all nodes in* SG.
2. *Edges connecting two vertices are associated with a single distance d.*
3. *If two* SG *nodes (n_i, n_i') map to the same tree vertex tn_i, they must co-exist in some tuple of* DB. *If there is an edge between two vertices (tn_i, tn_j) with distance d in* JKT, *then all pairs of corresponding nodes $n_i \in tn_i$ and $n_j \in tn_j$ must be related at d in* SG. *If two vertices (tn_i, tn_j) are not directly connected in* JKT, *then for each pair of nodes $n_i \in tn_i$ and $n_j \in tn_j$, there must be a relationship in* SG *at distance equal to that of the path connecting tn_i and tn_j.* □

Figure 5 shows a JKT of an SG containing four nodes $n_{johny,olson}$, n_{keep}, $n_{eternal}$ and n_{love} of the KRG in Figure 4. This tree is a JKT because (1) each vertex covers some node(s) existing in the SG and all nodes appear in

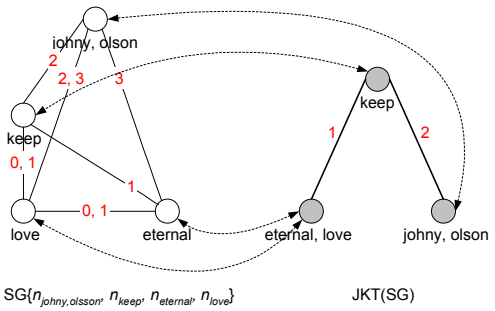


Figure 5: Mapping from *SG* to *JKT(SG)*

the tree vertices; (2) each edge has exactly one distance; (3) *Eternal* and *love* are in the same vertex $tn_{eternal,love}$ as they are related at distance 0 in the SG (they co-exist in t_7). Vertices $tn_{eternal,love}$ and tn_{keep} are directly connected by an edge with $d = 1$ and there is a relationship at distance 1 for pairs ($n_{eternal}$, n_{keep}) and (n_{love}, n_{keep}) in the SG. Similarly, vertices $tn_{johny,olson}$ and tn_{keep} are directly connected by an edge with $d = 2$ and there is a relationship at distance 2 for the pair ($n_{johny,olson}$, n_{keep}) in the SG. Finally, the length of the path between $tn_{eternal,love}$ and $tn_{johny,olson}$ is 3, and there is a relationship at distance 3 for both ($n_{eternal}$, $n_{johny,olson}$) and (n_{love}, $n_{johny,olson}$) in the SG.

Let q be a query that contains exactly the keywords represented by nodes in $SG \subseteq KRG$ and assume that the database corresponding to the KRG has query results. A *JKT(SG)* serves as a model for the construction of a *connection tree* of a result. In particular, a connection tree containing the query keywords can be built from JKT by adding intermediate nodes between JKT vertices connected through edges with distance greater than 1. For example, given $q = \{Olson, keep, eternal, love\}$, if we insert an intermediate node between tn_{keep} and $tn_{johny,olson}$ in the JKT in Figure 5, we derive a graph that has the same structure with the connection tree of Figure 2. This indicates that if KRG has an SG containing the query keywords, and there exists a *JKT(SG)*, it is likely that the database contains a query result. Following this fact, we give a definition of the Candidate Graph (CG) as follows.

DEFINITION 8. *Candidate graph, $CG(KRG, q)$*
Given a query q and a KRG, *$CG(KRG, q)$ is an* SG *of* KRG *satisfying the following properties*

1. *SG includes all nodes of* KRG *containing the query keywords, and only these nodes.*
2. *SG is complete (i.e., there is an edge between each pair of nodes).*
3. *There exists at least one $JKT(SG)$.* □

The KRG of Figure 4 has a candidate graph with respect to the query $q = \{Olson, keep, eternal, love\}$ because it includes a complete SG containing the four keywords and there is a JKT for this SG as shown in Figure 5. Next we discuss the relationship between the existence of candidate graphs in KRG and results in the corresponding database.

THEOREM 5.1. *If a database contains a result with all keywords of query q, then the corresponding* KRG *must have a candidate graph $CG(KRG, q)$.*

Proof: Assume a database containing a result $t_1 \bowtie t_2 \bowtie$ ... The existence of the result implies that the corresponding KRG has an SG, that contains a node for each query term. Since all keywords appear in $t_1 \bowtie t_2 \bowtie$ .., then all possible keyword pairs must be connected in SG. Therefore, SG satisfies the first two properties of a candidate graph. Next we show how to generate a JKT of SG.

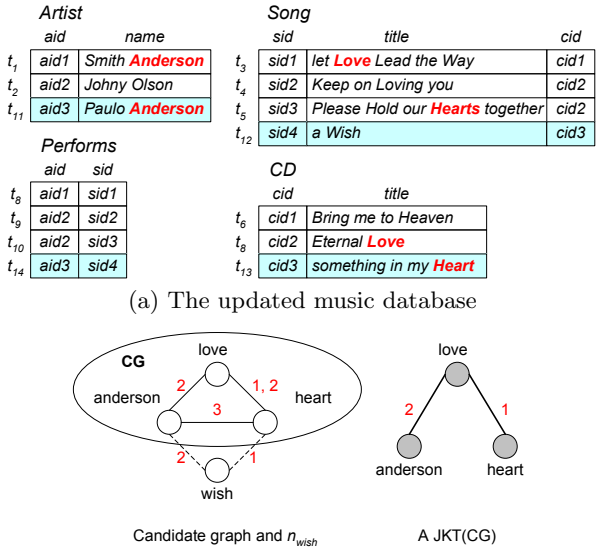
Based on the join sequence $t_1 \bowtie t_2 \bowtie$ .., we create a JKT using the following procedure. (i) Select a random tuple as a root. (ii) Pick another record, which can be joined with the first one, and insert it as a child of the root. (iii) While there are still unused tuples, pick a record t_i (among the remaining ones) that can join with an inserted tuple t_j and add t_i as a child of t_j. (iv) Set the distance of each link in the tree to 1. (iv) Replace each tuple with the set of query keywords inside. (v) For every record t_k that does not contain any keyword, remove t_k from the tree, connect its parent and its children, and increase the distance of these links by 1. The final output is a $JKT(SG)$. □

Note that a join sequence may create multiple $JKTs$ depending on the order of insertions in the tree. These $JKTs$ are equivalent in the sense that they correspond to the same result. Theorem 5.1 can be used to filter out databases that are guaranteed not to include results (i.e. databases not having a candidate graph). Although the existence of $CG(KRG, q)$ is a **necessary** condition in order for KRG to contain results, as shown in the following theorem, it is not **sufficient**. Furthermore, the proof of Theorem 5.2 provides important insight about the structure of *false positives* in *G-KS*, i.e., candidate graphs that fail to produce actual results.

THEOREM 5.2. *The existence of a candidate graph* $CG(KRG, q)$ *in* KRG *does not guarantee that the corresponding database has results for* q.

Proof: Let n_x be a node of $CG(KRG, q)$ representing a query keyword k_x, which is mapped to a vertex tn_x of a *JKT(CG)*; n_y is a KRG node for term k_y that does not belong to CG (i.e., k_y is not a query keyword). Assume that for each node $n_i \neq n_x, n_i \in CG$, if n_i is mapped to the same tree vertex tn_x as n_x, there exists a relationship between n_i and n_y at distance 0. If n_i is mapped to a tree vertex $tn_i \neq tn_x$, there exists a relationship between n_i and n_y at distance equal to the distance between tn_i and tn_x in the tree structure. In other words, if we replace n_x with n_y we obtain *JKT(SG')*, where $SG' = (CG \setminus n_x) \cup n_y$. We further assume that: (1) n_x and n_y appear at different tuples t_x and t_y that are not connected; and (2) there are two nodes n_i, n_j in $CG(KRG, q)$ related by a join through tuple t_y. Consequently, the actual join sequence forming the JKT may be due to SG'. In this case, there may not exist any result in the database containing all the keywords even though the KRG has a candidate graph. □

Figure 6(a) shows an updated instance of the music database of Figure 1, where there are new tuples t_{11} to t_{14}. These tuples indicate that artist *Paulo Anderson* (t_{11}) performs (t_{14}) a song *Wish* (t_{12}) in a CD entitled *Something in my heart* (t_{13}). Now, let us consider a query $q = \{Anderson, love, heart\}$. The KRG includes a candidate graph because the SG containing the query keywords is complete and has a *JKT(SG)* as shown in Figure 6(b). However, although $CG(KRG, q)$ exists, the database does not have a result for this query. Observe that any join sequence originating



(a) The updated music database

(b) CG and JKT

Figure 6: Candidate graph without result

from *Paulo Anderson* (t_{11}) does not contain *Love*, whereas any sequence from *Smith Anderson* (t_1) does not include *heart*. We demonstrate the reason for the failure by setting $k_x = love$ and $k_y = wish$ in the above proof. The vertices $tn_{Anderson}$, tn_{heart} corresponding to *Anderson* and *heart* are related with tn_{love} at distances 2 and 1 in JKT, respectively. In KRG, $n_{anderson}$, n_{heart} are connected with n_{wish} at the same distance. Furthermore, *wish* and *love* exist in different tuples that are not connected at any distance. This implies that there is not way to distinguish if $n_{anderson}$ and n_{heart} are connected through n_{love}, or through n_{wish} (as is the case here) because KRG (and consequently *JKT(SG)*) does not represent the paths between nodes, but only their distances. We refer to $KRGs$ that contain candidate graphs, but not query results, as *false positives*.

Even though a candidate graph does not provide certainty, it does generally indicate a high probability of the database having results. Let us consider a query with κ keywords and assume that each keyword is represented by a node in KRG. There are totally C_κ^2 relationships between pairs of nodes in the SG containing the query keywords. To determine if the SG is a candidate graph, we need to find a *JKT(SG)*. The third condition of the JKT definition goes beyond the binary relationships in KRG and tests collectively all relationships as the whole. This is due to the fact that a test on a pair of vertices in the JKT involves both (i) checking their direct relationship, and (ii) relationships between vertices in the path connecting them. Therefore, for a false positive to occur, the node n_y of Theorem 5.2 has to satisfy $\kappa - 1$ relationships to other nodes (to form a complete graph) and also $C_\kappa^2 - (\kappa - 1)$ hidden relationships between the remaining nodes. In most cases this probability is low and decreases as the number of query keywords increases. On the other hand, recall that *M-KS* simply involves binary relationships and the chance of false positives increases with the query size.

5.2 Search for candidate graphs

Given a query q, *G-KS* retrieves the subgraph SG of KRG that contains all the query keywords and checks whether it

Algorithm 1 : *ValidateSG* (Graph SG)

```
1: Create an empty tree T, an empty stack S
2: Create a list L containing all nodes in SG
3: n_R = Remove a node from L
4: Set n_R as the root of T
5: Push (T, L) to S
6: while S is not empty do
7:    Pop out (T, L) from S
8:    if L is empty then
9:       return TRUE
10:   else
11:      n_x = Remove a node from L
12:      for each node n_i in T do
13:         T_1 = Add n_x to the same vertex as n_i in T
14:         if T_1 can lead to a JKT then
15:            Push (T_1, L) to S
16:         T_2 = Add n_x as a child of n_i in T
17:         if T_2 can lead to a JKT then
18:            Push (T_2, L) to S
19:         for each child n_c of n_i in T_2 do
20:            T_3 = Remove n_c from n_i and add it as a child
               of n_x in T_2
21:            if T_3 can lead to a JKT then
22:               Push (T_3, L) to S
23: return FALSE
```

is complete. If either keyword coverage, or completeness is violated, then SG cannot be a candidate graph, implying that the corresponding database cannot lead to any result. This is similar to the filtering mechanism of *M-KS*. However, whereas *M-KS* would consider successful all *SGs* that feature keyword coverage and completeness, *G-KS* provides an additional level of filtering by verifying whether SG is a candidate graph.

Algorithm 1, hereafter referred to as *ValidateSG*, determines whether a valid join keyword tree can be induced in SG. Since we only need a single JKT, we employ a depth-first traversal [5]. Specifically, *ValidateSG* initializes a list L containing all nodes in SG, randomly picks a node in L as the root of a tree T, and pushes a pair of (T, L) to a stack S (*lines 1-6*). Then, it iteratively pops out (T, L) from S until either S or L is empty (*lines 7-23*). If S is empty, no solution JKT exists, and hence SG is not a candidate graph. On the other hand, if L is empty, all nodes have been assigned to a JKT T, and SG is a candidate graph. In each of the repeated steps, *ValidateSG* selects a node n_x among the remaining in L and considers all positions for n_x at the existing tree T. In particular, if there are S_n vertices in T, there are $(3 \cdot S_n - 1)$ possible locations for n_x: (i) S_n cases for inserting n_x in the same vertex as an existing node n_i (*lines 14-16*), (ii) S_n cases of assigning n_x as a child of a existing vertex (*lines 17-19*) and (iii) $S_n - 1$ cases of inserting n_x between two parent-child vertices in the tree (*lines 21-23*). If after the addition of n_x, the tree can lead to a JKT i.e. vertices in the tree satisfy the third condition of a JKT, T and L are pushed to the stack S.

For simplicity, the pseudo-code presents only the basic functionality of *ValidateSG*. In addition, we employ two strategies to speed up the process of finding a JKT. First, instead of selecting nodes randomly, we sort them by their total number of different distances with respect to other nodes in SG. Then, at each step, we select the node with the minimum number of distances. This strategy reduces the possible valid positions for a new node (to be inserted in the tree) and minimizes the number of potential *JKTs* in the stack. Consequently, if the graph has no JKT, the process can stop after exploring a small number of potential trees. Furthermore, the first nodes in the list form a stable solution; therefore, when backtracking is necessary, the number of backtracking steps is relatively low. Second, instead of trying all possible positions when inserting a node into a tree, we only attempt to add n_x around the existing node n_y with the smallest distance to n_x. Recall that for the new tree to satisfy the requirements of the JKT definition, the distance between n_x and n_y has to exceed the distance between n_x and every intermediate node n_z in the path connecting n_x and n_y. This requirement cannot be satisfied if n_x is attached to n_z.

As an example of the optimizations, consider searching for a JKT in the graph of Figure 5. Each of n_{keep}, $n_{eternal}$, $n_{johny,olson}$ has 4 possible distances to other nodes, while n_{love} has 6. We insert the nodes in a priority list L_P = (n_{keep}, $n_{eternal}$, $n_{johny,olson}$, n_{love}) (the first three nodes are ordered at random). Given L_P, the root of the tree is n_{keep}. Next, $n_{eternal}$ is inserted as a child of n_{keep}. Note that $n_{eternal}$ cannot be added to the same vertex as n_{keep} because they are not related at distance 0 (therefore the resulting tree could not lead to JKT). During the insertion of $n_{johny,olson}$, the second optimization considers the positions around n_{keep} (i.e., the node with the minimum distance to $n_{johny,olson}$). Observe that if we add $n_{johny,olson}$ as a child of $n_{eternal}$, the distance between $n_{johny,olson}$ and n_{keep} must be greater than the distance between $n_{johny,olson}$ and $n_{eternal}$ violating the JKT conditions. Thus, $n_{johny,olson}$ can only be appended as a child of n_{keep}. Finally, n_{love} is inserted in the same vertex as $n_{eternal}$, yielding the JKT of Figure 5.

5.3 Selection of top-K databases

We describe the selection of the top-K databases, considering both AND and OR semantics. AND semantics require each result to contain all query keywords. According to OR semantics a result may include only a subset of the keywords. Let κ be the term cardinality of a query q, and $|cg|$ be the largest number of query keywords that yield a candidate graph in the KRG of a database DB. If $|cg| < \kappa$, then according to Theorem 5.1, DB cannot produce any result for q based on AND semantics, and can be safely eliminated. On the other hand, for OR semantics, DB may still constitute a candidate for search. For two databases DB_1 and DB_2, if $|cg_1| > |cg_2|$, then DB_1 is, in general, expected to be a better candidate than DB_2. However, in certain cases DB_2 may rank higher, if the query keywords have large weights in its KRG, and occur in tuples close to each other.

The basic functionality of database selection is shown in Algorithm 2, hereafter referred to as *SelectDB*. *SelectDB* uses a set $S_{|cg|}$ to store the ids of all databases that contain candidate graphs with $|cg|$ keywords ($|cg| \leq \kappa$). The individual sets for various values of $|cg|$ are maintained in a set list L_{DB}. The algorithm first finds databases containing candidate graphs with all query keywords. The ids of these databases are inserted into S_κ and L_{DB}. If the query is based on AND semantics, *SelectDB* terminates and returns

[5]There may exist numerous *JKTs* because two nodes can be connected at different distances, each leading to a different JKT.

Algorithm 2 : *SelectDB*(Query q)

```
1: Create an empty list L_DB
2: |cg| = κ = number of keywords in q
3: repeat
4:    Create an empty set S_|cg|
5:    for all databases DB except for those in L_DB do
6:       if there is a candidate graph of |cg| keywords in DB
         then
7:          Add DB to S_|cg|
8:    Add S_|cg| to L_DB
9:    if "AND semantics" then
10:      break
11:   |cg| = |cg| - 1
12: until termination condition
13: return L_DB
```

L_{DB} to a scoring mechanism (to be discussed shortly).

In case of OR semantics, the termination condition is flexible. At one extreme, a user may want to rank all databases containing one or more keywords. At the other extreme, another user may wish to rank only the K databases that contain the maximum number of keywords. To accommodate this flexibility, *line 12* does not specify an explicit termination condition [6]. Assume that after finding databases with $|cg| = \kappa$, we want to retrieve databases with $|cg| = \kappa - 1$. *SelectDB* derives κ new queries of length κ-1, by removing each keyword in turn from q (for simplicity, the details of this process are omitted from the pseudo-code). All databases, except for those already in L_{DB}, are evaluated against the new queries, and those containing candidate graphs are inserted into $S_{\kappa-1}$ and L_{DB}. This process is repeated for κ-2 and so on, until the termination condition. The list L_{DB} contains the candidate databases in decreasing order of their $|cg|$, for facilitating ranking based on the size of the candidate graphs.

Given the wealth of information maintained by the *KRG*, *G-KS* can integrate a wide variety of scoring mechanisms. Adopting an IR-based methodology, we calculate the similarity between a database and a query as:

$$Score(DB, q) = \sum_{\{k_i,k_j\}\subseteq q, i\neq j} Score(k_i, k_j) \quad (1)$$

where $\{k_i, k_j\}$ is a pair of terms in q, and $Score(k_i, k_j)$ is their accumulated score calculated as:

$$Score(k_i, k_j) = w_i \cdot w_j \cdot \sum_d w_{ij}(d) \quad (2)$$

Recall from Section 3, that w_i and w_j are the weights of nodes n_i and n_j, representing terms k_i and k_j, respectively, while $w_{ij}(d)$ is the weight of the edge connecting n_i and n_j at distance d. *G-KS* computes the score of each database in L_{DB} based on the above equations and returns the top-K with the highest scores. Note that, for AND semantics, L_{DB} may contain contain fewer than K databases. In such cases, the benefits of *G-KS* are even more important as it saves redundant search in numerous databases that cannot contain any results.

5.4 Comparison of *G-KS* and *M-KS*

In this section, we discuss the differences between *G-KS* and *M-KS*, assuming, for consistency, that *M-KS* uses the SUM function [7] to calculate the similarity score between a database and a query. Specifically, $Score(DB, q)$ is also given by Equation 1, but the score of a keyword pair is:

$$Score(k_i, k_j) = \sum_d \frac{pf_{SUMij}(d)}{d+1} \quad (3)$$

where $pf_{SUMij}(d)$ is the total number of occurrences of the pair (k_i, k_j) at distance d calculated as:

$$pf_{SUMij}(d) = \sum_{t_x,t_y} pf_{ij}(t_x, t_y, d) \quad (4)$$

M-KS considers all keyword terms to be equally important, and simply counts the number of term co-occurrences to compute the weights of distance relationships in the matrix. Therefore, it cannot deal with the problem of popular terms, mentioned in Section 2.1. For instance, a database that contains frequent co-occurrences for a pair of query terms would achieve a large score, although these co-occurrences do not have high discriminating value (indeed, because of their frequency). In contrast, *G-KS* normalizes the weights of nodes and edges, solving the problem of popular terms.

The second difference between the two methods is due to the use of compound nodes in *KRG*. Compound nodes represent multiple terms that occur a single time in the database, and at the same tuple. This concept leads to effective compression of the *KRG*, since all terms in a compound node share the same set of distance relationships. In contrast, *M-KS* explicitly represents the distance relationship between all pairs of terms, without utilizing any compression mechanism. The resulting *KRM* has more entries than the corresponding *KRG*, consumes more space and incurs higher cost during query processing.

The third, and most important, difference regards the effectiveness of pruning. *M-KS* simply uses the binary relationships of queried keywords in *KRM* to calculate similarity scores. Consequently, it incurs a large number of false positives for queries with more than two keywords. In contrast, the concept of the candidate graph allows *G-KS* to rise above binary relationships, and treat the query terms holistically. As we show in the next Section, this leads to significant gains in terms of precision and recall.

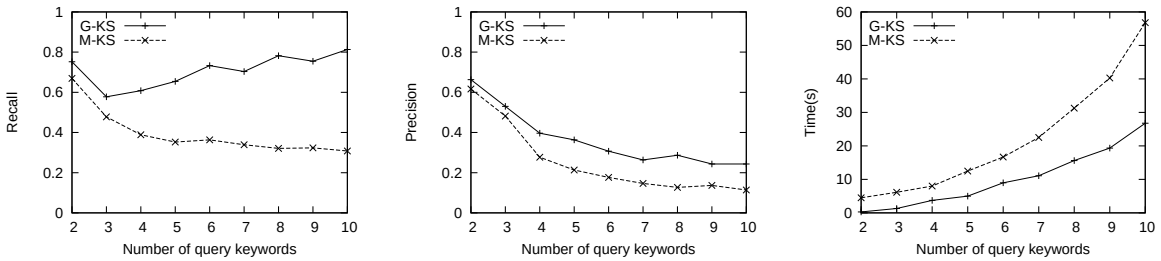
6. EXPERIMENTAL EVALUATION

We assume a client-server architecture where the server stores summaries of several databases. A client issues a query q to the server, which returns the part of the summary that is relevant to q, i.e., the weights of the query terms and their distance relationships. Given this information, the client selects the top-K databases locally[8]. In addition to responding to queries, the server has to construct and maintain the summaries in the presence of database updates. The server uses MySQL 4.1.7 to store and retrieve the summaries. Specifically, each KRG is represented by: (1) a *node* table where a tuple (n_i, w_i) signifies that the weight

[6]Details about the concrete implementation are discussed in Section 6.

[7][27] suggests using the SUM, PROD, MAX or MIN functions for combining the pairwise scores. PROD and MAX have a similar effect to SUM, while MIN cannot capture OR semantics.

[8]We chose this architecture because of its flexibility, e.g., each client may implement the scoring mechanisms of its choice. Alternatively, query processing can take place at the server.



(a) Recall (b) Precision (c) Query response time

Figure 7: Recall, precision, and query response time versus κ

Table 2: Pre-processing cost

	Time (Hours)	Space (MB)
M-KS	7.26	438.12
G-KS	4.97	362.59
Improvement	31.54%	17.24%

Table 3: Query parameters

Parameter	Default value	Range of values
δ	4	1 - 4
κ	5	2 - 10
K	3	2 - 10

of node n_i is w_i, (2) a *term* table where a tuple (k_j, n_i) represents that term k_j is mapped to node n_i, and (3) an *edge* table, where a tuple (n_i, n_j, d, w) indicates that nodes n_i and n_j are connected at distance d; w is the weight of the distance relationship. The client is an Intel Pentium 4 CPU 2.4 GHz and the server is an Intel Xeon MP CPU 3.00 GHz. The simulator is implemented using Java JDK 5.0.

Due to the unavailability of multiple real databases, we follow a methodology similar to [27]. Specifically, we decompose DBLP [7] into distinct databases according to bibliography types such as *articles*, *proceedings*, *books*. Since some of the resulting databases (e.g., *articles*, *proceedings*) are rather large, we decompose them further, to derive a total of 81 databases, each containing in the order of 50,000 records residing in 4-5 tables. We evaluate *G-KS* against *M-KS* [27] as there is no other competitor in the literature. The client-server framework remains the same, except that the server maintains *KRMs* instead of *KRGs*. *G-KS* applies the compression technique of Section 3.3 to reduce the *KRG* size. Section 6.1 compares the two methods on the preprocessing overhead. Section 6.2 focuses on the effectiveness and efficiency of query processing. Section 6.3 investigates the effect of delayed updates on the data summaries.

6.1 Pre-processing overhead

Table 2 illustrates the average construction time and space consumption per summary, assuming that the maximum distance relationship δ preserved in each summary is 4. *G-KS* is faster (31.54%) and requires less disk space (17.24%) than *M-KS* because of the compression of *KRGs*. For instance, given a database of 22,423 distinct terms and 53,214 tuples, *G-KS* creates a total of 20,308,927 distance relationships in the compressed *KRG*, whereas *M-KS* creates a total of 24,661,881 relationships in the *KRM*.

6.2 Query processing

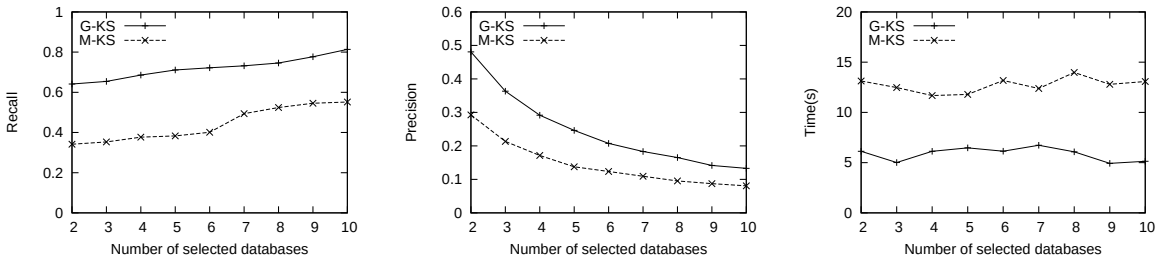
Since the probability for two random keywords to have an interesting association is very low, we generate queries containing terms that are pair-wisely related in at least some *KRGs*. Furthermore, we assume OR semantics, so that a result may miss some query keywords. *G-KS* uses the techniques of Section 5.3 to rank the databases. The termination condition of Algorithm 2 is: $|L_{DB}| \geq K$, i.e., *SelectDB* stops at a value of $|cg|$ such that the number of databases with candidate graphs is at least K. We also apply *SelectDB* for *M-KS* since the original version of the method cannot handle OR semantics. *M-KS* implements the scoring functions of Section 5.4.

The experiments focus on the efficiency and the effectiveness of the methods. Efficiency refers to the query response time, which is the duration from the instance that the client sends a query q to the server, until the time it produces the top-K databases for q. This includes (1) the retrieval of summaries at the server, and the extraction of the relevant information for q, (2) the network latency for transmitting this information to the client, and (3) the processing cost at the client. The network latency is very small and similar in both *G-KS* and *M-KS*; thus, the response time and the performance difference between the techniques is dominated by the other two factors.

Effectiveness refers to the concepts of precision and recall. In order to measure effectiveness, we use a brute-force (BF) method that, given q, receives the top-50 results from every database using the method of [17]. Let M be the maximum number of keywords in any result ($M \leq \kappa$). BF selects the top-K databases based on the cardinality of the results containing M keywords. For instance, if DB_1 returns 10 results with M=5 keywords, and DB_2 returns 20 results with 4 keywords, then DB_1 obtains a higher score than DB_2. Let $K_{BF}(M)$, $K_{GKS}(M)$ be the total number of results with M keywords in BF and *G-KS*, respectively. Then, *recall* is defined as $K_{GKS}(M)/K_{BF}(M)$. *Precision* is the ratio of the number of selected databases having results with M keywords in *G-KS* over K. The recall and precision of *M-KS* are defined in the same way.

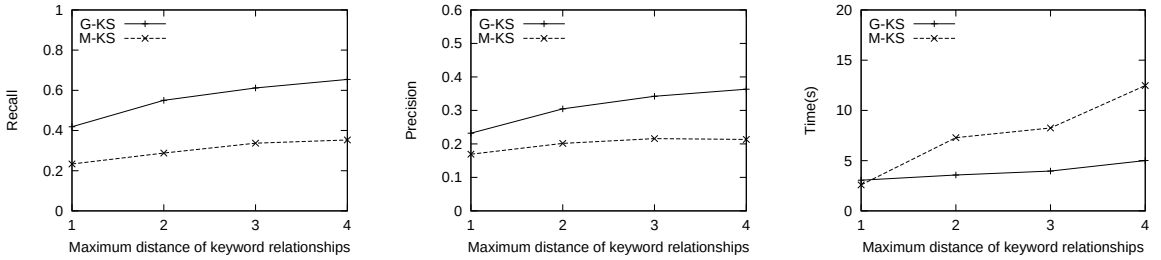
The experiments involve three parameters: the query length κ, the maximum distance δ between any two terms in the summary, and the number K of selected databases. Table 3 summarizes the default and range of values of these parameters. For each experiment we vary one parameter, while setting the other two to their default values. A number of 100 queries are executed and the displayed results are the average results from executing these queries.

Figure 7 evaluates recall, precision and response time as a function of κ. When the number of query keywords increases, the recall of *M-KS* drops due to false positives incurred by the binary filtering mechanism. On the other hand, *G-KS* checks relationships holistically through *Val-*



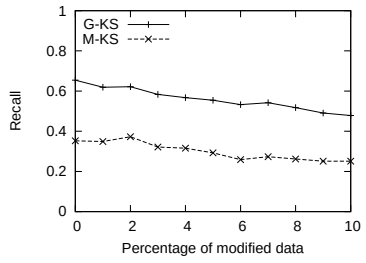
(a) Recall (b) Precision (c) Query response time

Figure 8: Recall, precision and query response time versus K

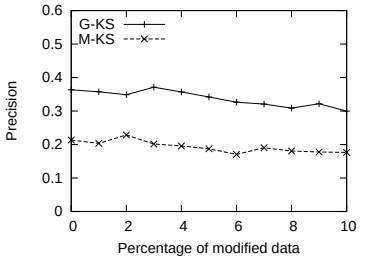


(a) Recall (b) Precision (c) Query response time

Figure 9: Recall, precision and query response time versus δ



(a) Recall



(b) Precision

Figure 10: Recall and precision versus data modification

idateSG and its recall grows with κ. Note that for large values of κ, there may be less than K databases with the maximum number of keywords, leading to relatively low precision. However, the high value of recall implies that *G-KS* obtains most of these databases. In addition to effectiveness, *G-KS* outperforms *M-KS* in terms of the efficiency. This result may initially appear counter-intuitive since *G-KS* performs the additional step of filtering through candidate graphs. However, the total processing cost is dominated by the retrieval of keyword relationships in the summaries. Since each *KRG* is significantly smaller that the corresponding *KRM*, the retrieval cost of *G-KS* is lower.

Figure 8 measures the effect of the number K of selected databases. As expected, the recall increases with K since more databases are likely to produce more results. On the other hand, the precision drops since a larger percentage of these results is irrelevant. Again *G-KS* has an important advantage over *M-KS* on both measures. The total processing cost is insensitive to the value of K, because the server transmits the same amount of information to the client, independently of the number of databases to be selected. Therefore, the overhead of summary retrieval, which is the dominant factor on the response time, does not change.

Figure 9 studies the effect of δ. A large δ implies detailed information in the summary. *G-KS* utilizes the additional information to improve both precision and recall. On the other hand, the same is not necessarily true for *M-KS*, as more relationships at higher distances may increase the number of false positives, misleading the system to over-estimate the usefulness of a database. The efficiency drops because the summary size is proportional to δ and larger summaries lead to longer retrieval times for keywords relationships.

6.3 Effect of delayed updates

As discussed in Section 4.2, to avoid the cost of frequent revisions, the *KRG* of a database may be updated in batches, instead of after every record insertion and deletion. This experiment evaluates the effectiveness as a function of the number of updates that have not been reflected in the summary. An update deletes a random record from one database, and inserts it in another one. Figure 10 shows precision and recall versus the percentage of the database size that has been modified. As expected, the effectiveness of both *M-KS* and *G-KS* decreases with the percentage of the modified data. Note that the results of *G-KS* when 10% of the data

have been modified, are better than those of *M-KS* before any updates.

7. CONCLUSION

This paper proposes *G-KS*, a method that selects the top-K databases for processing a relational keyword search query. *G-KS* summarizes each database as a keyword relationship graph, where nodes correspond to terms, and edges capture distance relationships. Both nodes and edges are weighted according to state-of-the-art IR techniques in order to support a variety of scoring functions. Based on the *KRG*, *G-KS* applies an intricate algorithm to identify and eliminate non-promising databases. As opposed to the previous work that is based only on binary relationship between terms, *G-KS* considers all query keywords as a whole in order to minimize the number of false positives. An extensive experimental evaluation confirms the superiority of *G-KS* in terms of effectiveness, efficiency, processing and pre-processing overhead. [22]

8. REPEATABILITY ASSESSMENT RESULT

Repeating the full experiments of this paper takes more than 40 days. Due to a lack of time, and to the authors' suggestion, the repeatability committee attempted to reproduce the experiments on a reduced version of 20 databases instead of 81 used in the paper. On this small database, the experiments described in Figures 7-9 ran successfully to completion. The experiment for Figure 10 was interrupted prior to completion due to lack of time. The observed results were compatible with the ones in the paper, while exhibiting important differences with the latter. The smaller scale is a satisfactory explanation for these differences. However, the repeatability committee does not feel confident to predict how the smaller-scale results would translate to the dataset provided in the paper. In summary, due to time constraints, the repeatability committee was not able to reproduce the experiments in this paper. Code and data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/

9. REFERENCES

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

[2] BestPeer. http://www.bestpeer.com.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of ICDE*, 2002.

[4] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *Proceedings of SIGIR*, 1995.

[5] G. Cao, J.-Y. Nie, and J. Bai. Integrating word relationships into language models. In *Proceedings of SIGIR*, 2005.

[6] S. Cohen. XSEarch: A semantic search engine for XML. In *Proceedings of VLDB*, 2003.

[7] DBLP. http://dblp.uni-trier.de.

[8] C. Fellbaum, editor. *Wordnet: An Electronic Lexical Database*. MIT Press, 1998.

[9] J. Gao, J.-Y. Nie, G. Wu, and G. Cao. Dependence language model for information retrieval. In *Proceedings of SIGIR*, 2004.

[10] L. Gravano, H. Garcia-Molina, and A. Tomasic. GlOSS: Text-source discovery over the internet. *ACM Transactions on Database Systems (TODS)*, 24(2):229–264, 1999.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *Proceedings of SIGMOD*, 2003.

[12] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searched on graphs. In *Proceedings of SIGMOD*, 2007.

[13] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proceedings of VLDB*, 2003.

[14] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proceedings of VLDB*, 2002.

[15] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of VLDB*, 2005.

[16] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *Proceedings of VLDB*, 2004.

[17] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proceedings of SIGMOD*, 2006.

[18] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *Proceedings of SIGMOD*, 2007.

[19] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: Top-k keyword query in relational databases. In *Proceedings of SIGMOD*, 2007.

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

[21] R. Nallapati and J. Allan. Capturing term dependencies using a language model based on sentence trees. In *Proceedings of CIKM*, 2002.

[22] S3: Scalable, Shareable and Secure P2P Based Data Management System. http://www.comp.nus.edu.sg/~s3p2p.

[23] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[24] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *Proceedings of ICDE*, 2007.

[25] S. K. M. Wong, W. Ziarko, V. V. Raghavan, and P. C. N. Wong. On modeling of information retrieval concepts in vector spaces. *ACM Transactions on Database Systems (TODS)*, 12(3):299–321, 1987.

[26] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proceedings of SIGMOD*, 2005.

[27] B. Yu, G. Li, K. Sollins, and A. K. H. Tung. Effective keyword-based selection of relational databases. In *Proceedings of SIGMOD*, 2007.

[28] B. Yuwono and D. L. Lee. Server ranking for distributed text retrieval systems on the internet. In *Proceedings of DASFAA*, 1997.