Finding Frequent Co-occurring Terms in Relational Keyword Search

Yufei Tao Jeffrey Xu Yu
Chinese University of Hong Kong

ABSTRACT

Given a set Q of keywords, conventional *keyword search* (KS) returns a set of tuples, each of which (i) is obtained from a single relation, or by joining multiple relations, and (ii) contains all the keywords in Q. This paper proposes a relevant problem called *frequent co-occurring term* (FCT) retrieval. Specifically, given a keyword set Q and an integer k, a FCT query reports the k terms that are not in Q, but appear most frequently in the result of a KS query with the same Q. FCT search is able to discover the *concepts* that are closely related to Q. Furthermore, it is also an effective tool for refining the keyword set Q of traditional keyword search. While a FCT query can be trivially supported by solving the corresponding KS query, we provide a faster algorithm that extracts the correct results without evaluating any KS query at all. The effectiveness and efficiency of our techniques are verified with extensive experiments on real data.

1. INTRODUCTION

Given a set Q of keywords, a *keyword search* (KS) query returns a set of tuples, each of which (i) is obtained from a single relation or by joining several tables, and (ii) contains all the keywords[1]. To illustrate, we use four tables whose schemas are shown in Figure 1, where the underlined are primary keys. Each arrow represents a primary-to-foreign key relationship. For example, `AUTHOR` $\rightarrow$ `WRITES` means that the primary key *A_id* of `AUTHOR` is referenced by the *A_id* in `WRITES`. Figure 2 demonstrates the partial content of each table. Given a set Q of keywords: {*Tony*, *paper*}, the KS query returns the result in Figure 3.

To understand the result, first notice that Figure 3 is actually the output of the natural join:

$\texttt{AUTHOR}_{A\_name\,=\,\text{Tony}} \bowtie \texttt{WRITES} \bowtie \texttt{PAPER}.$

[1]This is the AND semantic, as is the focus of this paper. The OR semantic has also been addressed by [11, 16], where a qualifying tuple only needs to cover at least one query keyword.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

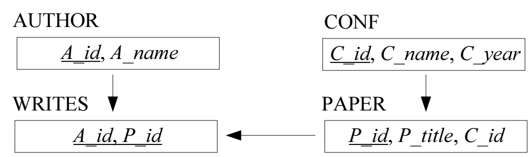


Figure 1: An example database schema

This expression is generated by the system automatically, as will be explained in Section 3. Second, each tuple in the result contains all the keywords in Q, treating the table name as an implicit keyword in every tuple of the table [16]. Third, the result is obtained with the smallest number of joins necessary. Specifically, the keywords *Tony* and *paper* are available only in `AUTHOR` and `PAPER`, respectively. Hence, every result tuple must combine a tuple in `AUTHOR` with one in `PAPER`, which, in turn, necessitates joins with `WRITES`.

Frequent Co-occurring Term Search. This paper proposes a new operator, *frequent co-occurring term* (FCT) retrieval, which adds a mining flavor to keyword search. Given a set Q of keywords, and an integer k, a FCT query returns the k most frequent terms in the result of a KS query with the same Q. For example, consider again the the result in Figure 3. Given the same Q and $k = 8$, a FCT query returns the 8 terms in Figure 4, which appear most frequently in Figure 3. Note that stop-words, such as "the", "of", etc. are excluded. Also excluded are the obvious noisy terms such as the table name `WRITES`. Furthermore, the keywords in Q are not considered either, since they must trivially appear in all result tuples. Finally, the standard word-stemming technique should be applied, so that words like "preservation" and "preserving" can be regarded as the same word.

Intuitively, a FCT query extracts the concepts that are most closely associated with the keyword set Q. For example, the terms in Figure 4 are indeed strongly related to Tony, since he has published primarily in two areas: (i) spatio-temporal (indexing and query processing) and (ii) privacy preserving data publication. Although the above discussion is based on the artificial example of Figure 2, similar observations indeed exist in the real world. For example, *query* and *spatio-temporal* are really the two most frequent terms in the titles of the papers by Tony; they appear 20 and 13 times, respectively.

Relevance to Traditional Keyword Search. The proposed FCT operator has a fundamental difference from the conventional KS queries: *FCT search extracts terms, while KS fetches tuples.* In particular, FCT search is different

AUTHOR

A_id	A_name
a_1	Tony
...	...

WRITES

	A_id	P_id
w_1	a_1	p_1
w_2	a_1	p_2
w_3	a_1	p_3
w_4	a_1	p_4
w_5	a_1	p_5
w_6	a_1	p_6
	...	...

symbols introduced for illustration purposes

PAPER

P_id	P_title	C_id
p_1	The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries	c_1
p_2	Time-Parameterized Queries in Spatio-Temporal Databases	c_2
p_3	The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries	c_3
p_4	Personalized Privacy Preservation	c_4
p_5	m-Invariance: Towards Privacy Preserving Re-publication of Dynamic Datasets	c_5
p_6	Preservation of Proximity Privacy in Publishing Numerical Sensitive Data	c_6
...	...	...

CONF

C_id	C_name	C_year
c_1	VLDB	2001
c_2	SIGMOD	2002
c_3	VLDB	2003
c_4	SIGMOD	2006
c_5	SIGMOD	2007
c_6	SIGMOD	2008
...	...	...

Figure 2: The table contents

A_id	A_name	P_id	P_title	C_id
a_1	Tony	p_1	The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries	c_1
a_1	Tony	p_2	Time-Parameterized Queries in Spatio-Temporal Databases	c_2
a_1	Tony	p_3	The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries	c_3
a_1	Tony	p_4	Personalized Privacy Preservation	c_4
a_1	Tony	p_5	m-Invariance: Towards Privacy Preserving Re-publication of Dynamic Datasets	c_5
a_1	Tony	p_6	Preservation of Proximity Privacy in Publishing Numerical Sensitive Data	c_6

Figure 3: Result of a KS query {*Tony*, *paper*}

term	*frequency*
spatio-temporal	3
query	3
privacy	3
preserve	3
tree	2
access	2
method	2
publish	2

Figure 4: The 8 most frequent terms in Figure 3

from *top-k KS* [11, 16, 17]. Given a keyword set Q and an integer k, a top-k KS query finds the k tuples (in the result of a normal KS query) most relevant to Q. The relevance is calculated by treating each tuple as a small document, and then applying an IR-style or page-rank relevance function. Hence, in top-k KS, tuples are mutually independent, as the relevance of a tuple does not depend on the others. In contrast, a FCT query must view all tuples in a holistic manner in order to aggregate the frequency of a term. Note that the k terms produced by FCT retrieval do not necessarily appear in the k tuples fetched by top-k KS. The reasons are two-fold. First, in top-k KS, the relevance of a tuple is decided by the keywords in Q, and hardly reflects the frequencies of the terms outside Q. Second, a tuple itself being relevant to Q does not imply that the terms it contains are globally frequent in the query result.

The explorative nature of FCT search also makes it an effective tool for refining KS queries. For example, consider someone that is interested in the research of Tony, but is not familiar with the areas he has worked in. Running a simple KS query with $Q = \{$*Tony*, *paper*$\}$ would return too many tuples, one for each publication. With a FCT query, s/he would be able to identify important terms that can be added to Q to formulate a more selective KS query. As shown in Figure 4, such terms could be *spatio-temporal* and *query*. Thus, next s/he would execute a KS query $Q = \{$*Tony*, *spatio-temporal*, *query*, *paper*$\}$ to retrieve only the papers of Tony on spatio-temporal queries.

Contributions. This paper presents a systematic study on FCT retrieval. We first provide a formal formulation of the problem, and then, propose a fast algorithm to solve it. We show that a FCT query can be answered *without* performing all the joins needed by the corresponding KS query. For instance, the terms in Figure 4 can be derived without computing the tuples in Figure 3. We have experimentally evaluated our technique on a real dataset *IMDB*, which incorporates the information of over 800k movies and TV programs. Our results show that FCT retrieval is effective, by revealing many interesting observations. Furthermore, our FCT algorithm significantly outperforms the straightforward approach of evaluating the corresponding KS query completely, achieving a maximum speedup of 4.

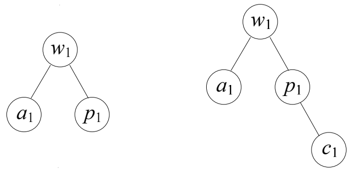
The rest of the paper is organized as follows. Section 2 formally formulates the problem of retrieving frequent co-occurring terms. Section 3 reviews the previous work on keyword search. Section 4 proposes an efficient FCT algorithm, and Section 5 extends the algorithm to the scenario where term appearances in various tables have different importance. Section 6 contains our experimental evaluation. Finally, Section 7 concludes the paper with directions for future work.

2. PROBLEM DEFINITION

We consider that the database has n tables $R_1, R_2, ..., R_n$, referred to as the *raw tables*. Their referencing relationships are summarized in a schema graph:

DEFINITION 1 (SCHEMA GRAPH). *The* schema graph *is a directed graph $\mathcal{G}$ such that (i) $\mathcal{G}$ has n vertices, corresponding to tables $R_1, ..., R_n$ respectively, and (ii) $\mathcal{G}$ has an edge from vertex R_i to vertex R_j ($1 \le i \ne j \le n$), if and only if R_j has a foreign key referencing a primary key in R_i.* □

For example, Figure 1 shows the schema graph of a database with $n = 4$ tables. Let Q be a set of m keywords $kw_1, ..., kw_m$. Each answer of the traditional keyword search is an MTJNT defined as follows:



(a) An MTJNT (b) Not an MTJNT

Figure 5: MTJNT illustration with Q = {*Tony*, *paper*}

Definition 2 (MTJNT). *A minimum total join network (MTJNT) is an undirected tree satisfying three requirements:*

- (join) *Each vertex is a tuple of a raw table. Let t and t' be any two adjacent vertices, and assume that they are in raw tables R and R' respectively. Then, R and R' must be connected in the schema graph, and $t \bowtie t'$ must belong to $R \bowtie R'$.*
- (total) *Every keyword in Q is contained in at least one vertex.*
- (minimal) *No vertex of the tree can be removed such that the remaining part is still a tree fulfilling the above requirements.* □

We assume that the name of a raw table R is an implicit term in each tuple in R. To illustrate MTJNTs, let us introduce some conventions for tuple referencing. For a tuple in tables `AUTHOR`, `PAPER`, `CONF` in Figure 2, we refer to it by its primary key, e.g., c_1 represents the first tuple in `CONF`. Given a tuple in `WRITES`, we denote it using the symbols w_1, w_2, ... shown in Figure 2, e.g., w_1 is the first tuple in `WRITES`. Figure 5a demonstrates an MTJNT for a query Q = {*Tony*, *paper*}. The tree in Figure 5b, however, is not an MTJNT, as it violates the minimal requirement in Definition 2. Namely, removal of vertex c_1 does not compromise the join- and total-requirements.

Definition 3 (Keyword Search). *Given a set Q of keywords and a number R_{max}, a* keyword search *(KS) query returns the set $KS(Q)$ of all possible MTJNTs that have at most R_{max} vertices.* □

The parameter R_{max} is introduced to prevent excessively large MTJNTs. For instance, with Q = {*Tony*, *paper*} and $R_{max} = 3$, $KS(Q)$ contains 6 MTJNTs, each of which is translated to a different tuple in Figure 3. In particular, the MTJNT in Figure 5a belongs to $KS(Q)$, and depicts the first tuple in Figure 3.

Let T be any MTJNT in $KS(Q)$. Given a term w, we use $count(T, w)$ to denote the number of occurrences of w in T, i.e., totally how many w are in the texts of the vertices of T. For example, let T be the MTJNT in Figure 5a. Then, $count(T, \textit{spatio-temporal}) = 1$ and $count(T, \textit{privacy}) = 0$, since the first tuple in Figure 3 contains one occurrence of *spatio-temporal* but no *privacy*.

Equipped with function $count(.,.)$, the total *frequency* of w in all MTJNTs can be obtained as:

$$freq(Q, w) = \sum_{\forall T \in KS(Q)} count(T, w). \qquad (1)$$

Now we are ready to define frequent co-occurring term retrieval:

Definition 4 (FCT search). *Given a set Q of keywords, a number R_{max}, and an integer k, a* frequent co-occurring term *(FCT) query returns the k terms with the highest frequencies among all terms that (i) are not in Q, and (ii) in the result of a KS query with the same Q and R_{max}.* □

Optionally, a user, who has some knowledge of the schema graph, may require that all the terms reported should appear in a particular set of relations.

As an example, given the database in Figure 2 and a query Q = {*Tony*, *paper*} and $R_{max} = 3$, a FCT query with $k = 4$ reports terms *spatio-temporal*, *query*, *privacy*, *preserve*, because they have the highest frequency 3 (see Table 4) among all terms in Figure 3 except *Tony* and *paper*.

As explained in Section 1, the motivation of FCT retrieval is to discover the concepts that best describe the characteristics of the query keyword set Q. Since it is based on term matching, standard pre-processing is needed to increase the accuracy. First, all the *stop-words* (i.e., common words such as "of", "is", etc.), noisy terms (i.e., words without significant meanings), and numerical data are excluded from consideration. Second, words with the same root (e.g., "preserving" and "preservation") should be counted as an identical word, as can be achieved through word-stemming.

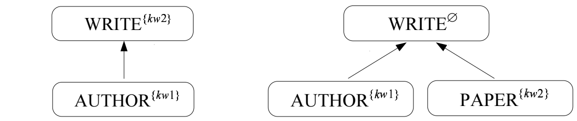
3. RELATED WORK

The previous works on relational keyword search can be divided into two categories, depending on whether they retrieve MTJNTs based on *candidate networks* (CN) or *data-graph traversal*. In the sequel, we outline their central ideas of both categories. Our discussion focuses on relational database (as is the topic of this paper). Nevertheless, at the end of the section, we will briefly survey relevant works on other types of data.

Methods Based on Candidate Networks. Keyword search (KS) aims at offering greater convenience to users in inquiring the database. Compared to conventional SQL queries, however, the convenience of KS is at the cost of higher complexity in query processing. As explained in the sequel, a major complication arises from the fact that MTJNTs may be produced from numerous different joins, depending on the distribution of the keywords in the raw tables.

Consider a KS query with a keyword-set Q. Given a raw table R and a subset S of Q, let R^S be the set of the tuples in R that (i) contain all the keywords in S, but (ii) do not include any keyword in $Q - S$. For example, consider the tables in Figure 2 and a KS query with a set Q of two keywords: kw_1 = *Tony*, kw_2 = *paper*. Then, $\texttt{AUTHOR}^{kw_1}$ includes all tuples that have *Tony* but not *paper*. As a special case, when $S = \emptyset$, R^S is the set of tuples in R that do not contain any keyword in Q at all. In general, for any non-empty S, R^S is called a *non-free tuple-set*. Otherwise (i.e., $S = \emptyset$), R^S is a *free tuple-set*.

Before accessing the underlying tables, the database must enumerate all the possible algebra expressions that may produce MTJNTs. The simplest expression is $\texttt{AUTHOR}^{\{kw_1, kw_2\}}$, namely, if a tuple in `AUTHOR` contains both *Tony* and *paper*,



(a) For expression E_1 (b) For expression E_4

Figure 6: Candidate network examples

the tuple itself constitutes an MTJNT. By the same reasoning, the other one-table expressions yielding MTJNTs are $\texttt{WRITES}^{\{kw_1,kw_2\}}$, $\texttt{PAPER}^{\{kw_1,kw_2\}}$, and $\texttt{CONF}^{\{kw_1,kw_2\}}$. There are more MTJNT-expressions involving two tables, for example:

$$\texttt{AUTHOR}^{\{kw_1\}} \bowtie \texttt{WRITES}^{\{kw_2\}}, \quad (E1)$$
$$\texttt{AUTHOR}^{\{kw_2\}} \bowtie \texttt{WRITES}^{\{kw_1\}}, \quad (E2)$$
$$\texttt{WRITES}^{\{kw_1\}} \bowtie \texttt{PAPER}^{\{kw_2\}}, ..., \quad (E3)$$

to list just a few. There exist even more MTJNT-expressions with three tables:

$$\texttt{AUTHOR}^{\{kw_1\}} \bowtie \texttt{WRITES}^{\emptyset} \bowtie \texttt{PAPER}^{\{kw_2\}}, \quad (E4)$$
$$\texttt{AUTHOR}^{\{kw_2\}} \bowtie \texttt{WRITES}^{\emptyset} \bowtie \texttt{PAPER}^{\{kw_1\}}, \quad (E5)$$
$$\texttt{WRITES}^{\{kw_1\}} \bowtie \texttt{PAPER}^{\emptyset} \bowtie \texttt{CONF}^{\{kw_2\}}, ..., \quad (E6)$$

Similarly, we can create a large number of MTJNT-expressions involving four tables. To avoid excessive expressions, a common approach [11, 12, 16, 17, 18] is to place an upper bound R_{max} on the number of tuple-sets in an expression. For example, with $R_{max} = 3$, it is not necessary to examine expressions with more than 3 tuple-sets.

An MTJNT-expression can be converted to a *candidate network* (CN). Specifically, given such an expression E, a CN is a *directed* tree where (i) a vertex corresponds to a (non-free or free) tuple-set in E and (ii) an edge between two vertices indicates that the two tuple-sets should be joined in evaluating E, and the edge's direction follows the direction of the corresponding edge in the schema graph. For example, Figure 6a (6b) presents the CN of the MTJNT-expression E_1 (E_4) shown earlier.

Hristidis and Papakonstantinou [12] develop an algorithm for generating all the candidate networks efficiently, subject to the upper bound R_{max}. This algorithm is deployed as the first step by all KS solutions [11, 12, 16, 17, 18] based on CNs. As a second step, a KS algorithm executes all the CNs (a.k.a MTJNT-expressions) to produce the MTJNTs. Note that a CN may not necessarily return any result. For example, $\texttt{AUTHOR}^{\{Tony,\ paper\}}$ is empty because no tuple in AUTHOR contains both *Tony* and *paper* simultaneously. The simplest approach of CN evaluation is to perform an SQL query for each CN. Various optimizations are possible for reducing the computation time. For example, many CNs may share common subexpressions, which, therefore, only need to be evaluated only once [12, 18]. Furthermore, if the goal is to report only the top-k MTJNTs (according to a certain scoring function) [11, 16, 17], the processing can be accelerated using the well-known *thresholding* technique [7] or a *skyline-sweep* approach proposed in [17].

Methods Based on Data Graphs. Based on their foreign-to-primary key relationships, the tuples in the raw tables can be connected into a *data graph*. Specifically, this is a directed graph, where (i) each vertex represents a tuple in a raw table, and (ii) there is an edge from tuple t to t' if and only if t' has a foreign key referencing the primary key of t. To illustrate, Figure 7 shows the data graph resulting from the database in Figure 2. Following the naming convention in Figure 5, for each tuple in AUTHOR, PAPER, CONF, we label its vertex with its primary key, whereas, for tuples in WRITES, their vertices are labeled with symbols w_1, w_2, ... defined in Figure 2. There is an edge from, for example, a_1 to w_1 because tuple w_1 (first row of WRITES) references the primary key of tuple a_1 (first row of AUTHOR).

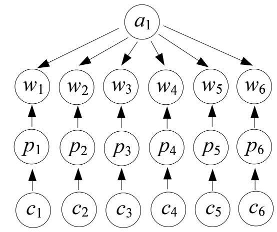


Figure 7: The data graph of the database in Figure 2

Given a set Q of keywords, the MTJNTs can be found by traversing the data graph. Next we explain the *backward* [4] strategy, which is the foundation of other more complex approaches [10, 14]. Let Q contain two keywords $kw_1 = Tony$ and $kw_2 = paper$. *Backward* first fetches the set S_1 (S_2) of tuples whose texts contain kw_1 (kw_2). In Figure 7, $S_1 = \{a_1\}$ and $S_2 = \{p_1, p_2, ..., p_6\}$, which can be easily obtained given an inverted index[2]. To obtain an MTJNT, *backward* picks two vertices from S_1 and S_2 respectively, and gradually expands their neighborhoods, until a common vertex is encountered in both neighborhoods. For instance, assume that we pick a_1 from S_1, and p_1 from S_2. For a_1, *backward* identifies its 1-edge neighborhood, i.e., the set $\{w_1, w_2, ..., w_6\}$ of vertices that can be reached from a_1 by crossing one edge. Similarly, the 1-edge neighborhood of p_1 is $\{w_1\}$ (c_1 is not included, because the edge between p_1 and c_1 is pointing at p_1). As w_1 appears in both the 1-edge neighborhoods of a_1 and p_1, *backward* outputs the MTJNT in Figure 5a, with w_1 being the root, and p_1, a_1 the leaves. In our example, all MTJNTs can be derived from 1-edge neighborhoods. In general, however, further neighborhood expansion may be necessary to guarantee no false miss.

All the algorithms [3, 4, 6, 10, 14] leveraging data-graphs deal with top-k KS, where the score of an MTJNT is calculated based on its tree structure, instead of purely from its texts. For example, the score of an MTJNT can be defined as the sum of the weights of all its edges. In this case, fast discovery of the top-k MTNJTs requires neighborhood expansions to be performed in a prioritized manner. This creates tremendous opportunities for optimization, aiming at expanding the most promising neighborhood earlier. Retrieval of the top-1 MTNJT is actually a classical steiner-tree problem, for which Ding et al. [6] give a dynamic-programming algorithm with good asymptotical performance. Kimelfeld and Sagiv [15] propose a theoretical algorithm for the general top-k version.

Whether edge weights are important is a crucial factor in choosing between KS solutions based on CNs (candidate network) and data graphs. In case edge weights must be considered, a data-graph method should be applied, because CN-algorithms are aware of only the foreign-to-primary connections at the schema level, but not at the tuple level. On the other hand, when edge weights are irrelevant, CN-

[2]For every term w in the database, the inverted index contains a list of tuples where w appears.

algorithms have better performance, since they can exploit the powerful execution engine of the database to extract multiple MTJNTs via a single join. For this reason, we also design our FCT algorithm following the CN-methodology.

Other Works. Keyword search has also been studied on non-relational data. In particular, considerable efforts [2, 9, 13, 21] have been made on XML documents. Recently, keyword-driven query processing is also introduced in spatial databases [8] and OLAP [20]. The above discussion focuses on centralized DBMS, whereas keyword search in distributed systems has also been investigated [19, 22].

4. THE STAR ALGORITHM

This section discusses the algorithmic issues in FCT search. Section 4.1 first provides the high-level description of the proposed algorithm. Then, Sections 4.2 and 4.3 explain the details of two major components.

4.1 High-level Description

A straightforward solution, referred to as *baseline*, to FCT retrieval is to first solve the corresponding KS query, and then extract the term frequencies. Specifically, given a set Q of keywords and an integer k, *baseline* applies a conventional KS algorithm (surveyed in Section 3) to retrieve the set $KS(Q)$ of all MTJNTs. Then, the algorithm computes the frequency *freq*(Q, w) of each term w by Equation 1, and reports the k most frequent terms.

Baseline incurs expensive cost because it *completely* evaluates all the joins necessitated by a KS query in order to obtain $KS(Q)$. While complete join-evaluation is mandatory for reporting MTJNTs, can it be avoided if our objective is to derive only the term frequencies? The answer is yes. Next, we design an algorithm *star* to acquire all term frequencies without producing the MTJNTs.

Star applies the methodology of candidate networks (CN) reviewed in Section 3. Specifically, given the query keyword-set Q, it employs the CN-generation algorithm in [12] to obtain all the CNs. Let us use h to represent the number of CNs, and denote them as CN_1, CN_2, ..., CN_h, respectively. Recall that, as explained in Section 3, a CN can be regarded as an algebra expression, which retrieves a set of MTJNTs. We deploy $MTJNT(CN_i)$ to denote the set of MTJNTs resulting from executing CN_i $(1 \le i \le h)$. Clearly, $MTJNT(CN_i) \cap MTJNT(CN_j) = \emptyset$ for any $1 \le i \ne j \le h$, that is, no MTJNT can be output by two CNs at the same time. It follows that

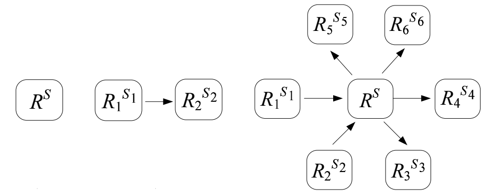
$$KS(Q) = \bigcup_{i=1}^{h} MTJNT(CN_i). \quad (2)$$

Let *freq-CN*(CN_i, w) be the total number occurrences of term w in all the MTJNTs of $MTJNT(CN_i)$, or formally:

$$\textit{freq-CN}(CN_i, w) = \sum_{\forall T \in MTJNT(CN_i)} \textit{count}(T, w). \quad (3)$$

where *count*(T, w), as defined in Section 2, is the number of occurrences of w in a single MTJNT T. Thus, the total frequency *freq*(Q, w) can be calculated as:

$$\textit{freq}(Q, w) = \sum_{i=1}^{h} \textit{freq-CN}(CN_i, w). \quad (4)$$



(a) (b) (c)

Figure 8: Star-CN examples

Therefore, the key to computing *freq*(Q, w) is to calculate *freq-CN*(CN_i, w) for a single candidate network CN_i.

The crucial observation behind the design of our *star* algorithm is that *freq-CN*(CN_i, w) can be calculated efficiently when CN_i is a *star-CN*:

DEFINITION 5 (STAR CANDIDATE NETWORK). *A* star candidate network *(star-CN) is a CN where a vertex, called the* root*, connects to all the other vertices, called the* leaves*.* □

Figure 8 demonstrates several example of star-CNs. Note that the simplest star-CN can have only a single tuple-set R^S (as explained in Section 3, R^S is the set of tuples in the raw table R that contain only the terms in S but not any term in $Q - S$, where Q is the query keyword-set). Also note that a star-CN can have any number of leaf tuple-sets. Furthermore, the edge directions can be arbitrary. For instance, in Figure 8c, some edges are pointing at the root R^S, while others away from R^S. In other words, R^S may reference the primary keys of some leaf tuple-sets, and meanwhile may be referenced by other leaves.

As elaborated in the next section, given a star-CN CN and a term w, *freq-CN*(CN, w) can be obtained at cost considerably lower than deriving the MTJNTs in $MTJNT(CN_i)$. This is why the proposed *star* algorithm is significantly faster than *baseline*. Apparently, when the schema graph itself is a star, all the CNs are definitely star-CNs. This makes our *star* algorithm especially suitable in data warehouse applications (where star schemas are common).

In case the schema graph is not a star, some CNs CN may not be star-CNs. In this case, we perform an interesting operation, called *starization*, to transform CN into a star-CN CN' that returns the same MTJNTs. Then, *star* proceeds normally with CN'. Intuitively, *starization* evaluates only the minimum set of joins to complete the conversion of CN into a star-counterpart. Note that those joins must be performed by the *baseline* approach anyway. In other words, *starization* carries out some of the work done by the *baseline* approach, but just enough to enable the application of *star*.

In the next subsection, we elaborate how to obtain the term frequencies from a star-CN. Then, Section 4.3 presents the details of *starization*.

4.2 Term Frequency Retrieval in a Star-CN

This section settles the following problem: given a star-CN CN, find the frequencies *freq-CN*(CN, w) of *all* terms w that appear in at least one MTJNT of $MTJNT(CN)$.

Let R^S be the tuple-set at the root of CN. Use l to denote the number of leaf tuple-sets in CN, and $R_i^{S_i}$ be the i-th leaf tuple-set $(1 \le i \le l)$. Deploy A_i to represent the set of joining attributes between R^S and $R_i^{S_i}$. Hence, conceptually R^S

has columns $\{A_1, A_2, ..., A_l, text\}$, where *text* incorporates all the attributes other than A_1, ..., A_l. In the same fashion, $R_i^{S_i}$ can be regarded to have a schema $\{A_i, text\}$. Following the convention of previous work [11, 12, 17, 16], we assume that the data of each tuple-set have been collected into a file. (This can be easily achieved with a single scan of all the raw tables. Furthermore, all the tuple-sets together consume exactly the same amount of space as the raw tables, because every tuple in a raw table belongs to precisely one tuple-set.)

We will use the running example in Figure 9 to illustrate our solution. Figure 9a gives the CN under consideration, assuming that the query keyword-set Q has three terms kw_1, kw_2, and kw_3. The root of CN is a free tuple-set $R^{\emptyset}$, i.e., no tuple in $R^{\emptyset}$ should contain any keyword in Q. All leaf tuple-sets are non-free. For example, in $R_1^{\{kw_1\}}$, each tuple must include only kw_1, but not kw_2 and kw_3. Notice that $R^{\emptyset}$ has two primary keys A_1 and A_2, referenced by $R_1^{\{kw_1\}}$ and $R_2^{\{kw_2\}}$ respectively, whereas $R^{\emptyset}$ itself references the primary key A_3 of $R_3^{\{kw_3\}}$.

Figures 9b-9e give the contents of the four tuple-sets. We use symbols α_1, α_2, ..., β_1, ..., γ_1, ..., δ_1, ... to facilitate tuple pinpointing. For instance, α_1 denotes the first tuple in $R_1^{kw_1}$. Note that some foreign-key values (e.g., x_3) of $R_1^{kw_1}$ are absent from the primary-key A_1 of $R^{\emptyset}$. This is reasonable because $R^{\emptyset}$ may not contain all the tuples in the raw table R (recall that $R^{\emptyset}$ is only a subset of R). Similar phenomena can be observed for the foreign-key values in other tables.

Figure 10 shows the result of completely evaluating CN. We refer to each tuple in the result as a *join tuple*, which is essentially a flat representation of an MTJNT in $MTJNT(CN)$. Symbols λ_1, λ_2, ... are added for tuple pinpointing. For example, tuple λ_1 is the join result of tuples α_3, β_1, γ_1, and δ_2. It is easy to verify that term w_1 occurs 16 times, i.e., *freq-CN*$(CN, w_1) = 16$. Similarly, *freq-CN*$(CN, w_2) = 13$, *freq-CN*$(CN, w_3) = 4$, and *freq-CN*$(CN, w_4) = 5$. In the sequel, we present the *star* algorithm that obtains these frequencies without producing the results in Figure 10.

	A_1	A_2	A_3	$R_1^{\{kw1\}}$.*text*	$R_2^{\{kw2\}}$.*text*	$R_3^{\{kw3\}}$.*text*	$R^{\emptyset}$.*text*
λ_1	x_2	y_1	z_1	kw_1, w_1, w_1, w_2	kw_2, w_4	kw_3, w_1	w_1, w_2
λ_2	x_2	y_1	z_1	kw_1, w_1, w_1, w_2	kw_2, w_2	kw_3, w_1	w_1, w_2
λ_3	x_2	y_1	z_1	kw_1, w_2	kw_2, w_4	kw_3, w_1	w_1, w_2
λ_4	x_2	y_1	z_1	kw_1, w_2	kw_2, w_2	kw_3, w_1	w_1, w_2
λ_5	x_2	y_1	z_1	kw_1, w_3	kw_2, w_4	kw_3, w_1	w_1, w_2
λ_6	x_2	y_1	z_1	kw_1, w_3	kw_2, w_2	kw_3, w_1	w_1, w_2
λ_7	x_4	y_3	z_2	kw_1, w_4	kw_2, w_3	kw_3, w_4	w_3

Figure 10: Result of complete evaluation of CN

Volume. Let t be a tuple in any tuple-set of CN. We define its *volume*, denoted as *vol*(t), as the number of join tuples determined by t. For instance, as mentioned earlier, join tuple λ_1 in Figure 10 is produced by tuple α_3 in Figure 9b (together with β_1, γ_1, δ_2). In fact, the volume *vol*(α_3) is 2, because α_3 also produces another join tuple λ_2. To see more examples, *vol*$(\beta_3) = 0$ (since β_3 does not produce any join tuple), and *vol*(γ_1) = *vol*$(\delta_2) = 6$ (since γ_1 determines λ_1, λ_2, ..., λ_6, and so does δ_2).

The central idea underlying *star* is that, *once we have obtained the volumes of the tuples in each tuple-set, we can precisely calculate the number of occurrences of any term.* Let us consider, for example, term w_1. This term appears twice in α_3. As α_3 yields *vol*$(\alpha_3) = 2$ join tuples, w_1 also appears twice in each of those two join tuples, thus contributing totally 4 occurrences. Similarly, w_1 also emerges once in γ_1 (or δ_2), and hence, has one occurrence in each of the *vol*$(\gamma_1) = 6$ (or *vol*$(\delta_2) = 6$) join tuples determined by γ_1 (or δ_2). This leads to another 12 occurrences of w_1, resulting in *freq-CN*$(CN, w_1) = 4 + 12 = 16$.

Motivated by this, *star* executes in two steps. The first phase, called the *volume step*, computes the volumes of all tuples in each tuple-set of CN. Then, the second phase, the *frequency step* calculates the frequency of each term.

Volume Step. This phase is further divided into the *leaf-stage* and the *root-stage*. The leaf-stage scans each leaf tuple-set $R_i^{S_i}$ $(1 \le i \le l)$ of CN once. The purpose is to prepare a *num*-array for the column A_i of $R_i^{S_i}$. For every value v in this column[3], *num*(v) equals the number of tuples in $R_i^{S_i}$ carrying v. For instance, for our running example in Figure 9, the leaf-stage outputs the *num*-arrays in Figure 11a. For example, *num*(x_1) equals 2, because in $R_1^{kw_1}$ two tuples have x_1 as their A_1-values. Notice that the *num*-array of each $R_i^{S_i}$ can be regarded as a compressed version of its column A_i. In particular, if a value v occurs many times in A_i, it is stored only once, but associated with its *num*-value.

The root-stage, on the other hand, reads the root tuple-set R^S once, and creates an *abridged* root tuple-set R_*^S, which has at most the same cardinality as R^S. Furthermore, this stage also obtains the volumes of all the tuples in every (root/leaf) tuple-set. At the beginning, the root-stage initiates a *vol*-array for each leaf tuple-set $R_i^{S_i}$ $(1 \le i \le l)$. For every value v in the column A_i of $R_i^{S_i}$, the array has an entry *vol*(v), which is initialized to 0. At the end of the root-stage, *vol*(v) will be identical to the volume of each tuple in $R_i^{S_i}$ whose A_i-value equals v. It suffices to keep only one volume for all tuples having the same A_i-value, as their volumes are equivalent.

Next, we process each tuple of the root tuple-set R^S in turn. Let $t = (v_1, v_2, ..., v_l, text)$ be the tuple being processed, where v_i $(1 \le i \le l)$ is its value on column A_i. Then, we check if t can be discarded. Specifically, as long as any v_i $(1 \le i \le l)$ does not exist in the *num*-array of leaf relation $R_i^{S_i}$, t does not produce any join result, and hence, can be safely eliminated. Otherwise (i.e., t cannot be discarded), we increase entry *vol*(v_i) in the *vol*-array of $R_i^{S_i}$ $(1 \le i \le l)$, using the data in the *num*-arrays of the other leaf tuple-sets. Formally, the update of *vol*(v_i) is by:

$$vol(v_i) = vol(v_i) + \prod_{j \ne i, 1 \le j \le l} num(v_j). \tag{5}$$

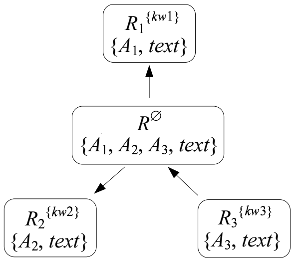
We also calculate the volume of t as

$$vol(t) = \prod_{j=1}^{l} num(v_j). \tag{6}$$

Finally, we write to the abridged root tuple-set R_*^S the tuple t, augmented with an additional field *vol*(t), and continue with the next tuple in R^S.

Let us demonstrate the root-stage with our running example. Recall that, in the leaf-stage, the *num*-arrays in

[3]In case A_i is a set of attributes, v is a vector.



(a) Star-CN CN

	A_1	text
α_1	x_1	kw_1, …
α_2	x_1	kw_1, …
α_3	x_2	kw_1, w_1, w_1, w_2
α_4	x_2	kw_1, w_2
α_5	x_2	kw_1, w_3
α_6	x_3	kw_1, …
α_7	x_4	kw_1, w_4

(b) $R_1^{kw_1}$

	A_2	text
β_1	y_1	kw_2, w_4
β_2	y_1	kw_2, w_2
β_3	y_2	kw_2, ...
β_4	y_3	kw_2, w_3

(c) $R_2^{kw_2}$

	A_3	text
γ_1	z_1	kw_3, w_1
γ_2	z_2	kw_3, w_4
γ_3	z_3	kw_3, ...

(d) $R_3^{kw_3}$

	A_1	A_2	A_3	text
δ_1	x_1	y_4	z_1	...
δ_2	x_2	y_1	z_1	w_1, w_2
δ_3	x_4	y_3	z_2	w_3

(e) $R^{\emptyset}$

Figure 9: A running example (kw_1, kw_2, and kw_3 are query keywords)

	x_1	x_2	x_3	x_4
num	2	3	1	1

	y_1	y_2	y_3
num	2	1	1

	z_1	z_2	z_3
num	1	1	1

(a) The *num*-arrays of $R_1^{kw_1}$, $R_2^{kw_2}$, and $R_3^{kw_3}$

	x_1	x_2	x_3	x_4
vol	0	0	0	0

	y_1	y_2	y_3
vol	0	0	0

	z_1	z_2	z_3
vol	0	0	0

(b) The *vol*-arrays at the beginning of the root-stage

	x_1	x_2	x_3	x_4
vol	0	2	0	0

	y_1	y_2	y_3
vol	3	0	0

	z_1	z_2	z_3
vol	6	0	0

(c) The *vol*-arrays after processing δ_2

	x_1	x_2	x_3	x_4
vol	0	2	0	1

	y_1	y_2	y_3
vol	3	0	1

	z_1	z_2	z_3
vol	6	1	0

(d) The *vol*-arrays after processing δ_3

	A_1	A_2	A_3	text	vol
δ_2^*	x_2	y_1	z_1	w_1, w_2	6
δ_3^*	x_4	y_3	z_2	w_3	1

(e) $R_*^{\emptyset}$

Figure 11: Illustration of the volume step

Figure 11a have been calculated. Before scanning the root tuple-set $R^{\emptyset}$, we initialize the *vol*-arrays of $R_1^{kw_1}$, $R_2^{kw_2}$ and $R_3^{kw_3}$ as in Figure 11b. We proceed to process the first tuple δ_1 of $R^{\emptyset}$, and discard it immediately because its A_2-value y_4 does not exist in the *num*-array of $R_2^{kw_2}$, implying that δ_1 does not produce any join tuple.

The next tuple processed is δ_2, which cannot be discarded because its A_1-, A_2-, and A_3-values x_2, y_1, z_1 all appear in the *num*-arrays. Thus, we update three entries in the *vol*-arrays, i.e., $vol(x_2)$, $vol(y_1)$, and $vol(z_1)$, resulting in the new *vol*-arrays in Figure 11c. Note that the updates are according to Equation 5. For example, $vol(x_2)$ is obtained as $num(y_1) \cdot num(z_1) = 2 \cdot 1 = 2$. Next, the volume of δ_2 is calculated with Equation 6, leading to $vol(\delta_2) = num(x_2) \cdot num(y_1) \cdot num(z_1) = 3 \cdot 2 \cdot 1 = 6$. Finally, we append t to the abridged root tuple-set $R_*^{\emptyset}$ along with its volume 6.

The last tuple δ_3 of R^S is processed in the same manner, yielding $vol(\delta_3) = 1$, and the final *vol*-arrays in Figure 11d. Some entries in the *vol*-arrays are 0, indicating that their corresponding tuples do not produce any join tuples. For example, in $R_1^{kw_1}$, tuples with A_1-value x_1 do not generate any join tuple. The root-stage terminates. Figure 11e gives the content of the current $R_*^{\emptyset}$. Notice that symbols δ_2^* and δ_3^* are introduced to enable tuple pinpointing. We formally summarize the entire volume-step in Figure 12.

Frequency Step. Once the tuple volumes have been computed, it is relatively easy to obtain the term frequencies. Towards this purpose, the frequency step performs one more scan on each leaf tuple-set and the abridged root tuple-set. Specifically, initially, *freq-CN*(CN, w) equals 0 for every w. Let t be a tuple in any of the tuple-sets mentioned earlier. When t is encountered, for each occurrence of a term w in t, we simply increase *freq-CN*(CN, w) by $vol(t)$. It remains to clarify how to retrieve $vol(t)$. If t is in the abridged root tuple-set R_*^S, $vol(t)$ is directly fetched along with t. Otherwise, assume that t comes from a leaf tuple-set $R_i^{S_i}$ (for some $i \in [1, l]$). We only need to obtain the A_i value v of t, and then, set the volume of t to the entry $vol(v)$ in the *vol*-array.

Algorithm *volume-step*

/* Input: star-CN CN with root tuple-set R^S and l leaf tuple-sets $R_1^{S_1}$, ..., $R_l^{S_l}$ */

1. scan each leaf tuple-set to prepare its *num*-arrays
2. initialize an all-zero *vol*-array for each leaf tuple-set
3. initialize an empty abridged root tuple-set R_*^S
4. while there are still un-processed tuples in R^S
5.   get the next un-processed tuple $t = (v_1, ..., v_l, text)$
       /* v_i $(1 \le i \le l)$ is the A_i-value of t */
6.   if all v_1, ..., v_l appear in the *num*-arrays
7.   for $i = 1$ to l
8. $vol(v_i) = vol(v_i) + \Pi_{j \neq i} num(v_j)$
9. $vol(t) = \Pi_{j=1}^{l} num(v_j)$
10. t^* = everything in t together with $vol(t)$
11.   add t^* to R_*^S

Figure 12: The volume step of algorithm *star*

To illustrate, let us calculate *freq-CN*(CN, w_1) in the example of Figure 9, from the *vol*-arrays (Figure 11d) and abridged root tuple-set $R_*^{\emptyset}$ (Figure 11e) returned by the volume step. At the beginning, *freq-CN*$(CN, w_1) = 0$. As (i) w_1 appears twice in α_3 and once in γ_1 and δ_2^* respectively, and (ii) $vol(\alpha_3) = 2$, $vol(\gamma_1) = 6$, $vol(\delta_2^*) = 6$, we have *freq-CN*$(CN, w_1) = 2 \cdot 2 + 1 \cdot 6 + 1 \cdot 6 = 16$. In particular, $vol(\alpha_3)$ is retrieved from the entry $vol(x_2)$ in the *vol*-arrays, where x_2 is the A_1-value of α_3. Likewise, $vol(\gamma_1)$ equals $vol(z_1)$ with z_1 being the A_3-value of z_1. Finally, $vol(\delta_2^*)$ is acquired directly from the abridged tuple-set $R_*^{\emptyset}$.

Discussion. The *star* algorithm described above is highly efficient. Specifically, regardless of the number l of leaf tuple-sets, *star* requires reading each tuple-set of CN only twice, and writing a tuple-set R_*^S once that is no larger than the root tuple-set R^S. This is much faster than the full evaluation of CN (i.e., returning all the join tuples as in Figure 10), which demands more passes on the participating tuple-sets. As mentioned before, the efficiency of *star* arises from the fact that it focuses on calculating only tuple vol-

Algorithm *frequency-step*

/* Input: the leaf tuple sets $R_1^{S_1}$, ..., $R_l^{S_l}$, the abridged root tuple-set R_*^S and the *vol*-arrays output by the volume-step */

freq-CN$(CN, w) = 0$ for all terms w
for each leaf tuple-set $R_i^{S_i}$ $(1 \leq i \leq l)$
    while there are still un-processed tuples in $R_i^{S_i}$
        get the next un-processed tuple $t = (v, \mathit{text})$
            for each occurrence of any term w in t
                freq-CN(CN, w) = *freq-CN*$(CN, w) + \mathit{vol}(v)$
while there are still un-processed tuples in R_*^S
    get the next un-processed tuple $t^* = (v_1, ..., v_l, \mathit{text}, \mathit{vol}(t^*))$
        for each occurrence of any term w in t^*
            freq-CN(CN, w) = *freq-CN*$(CN, w) + \mathit{vol}(t^*)$

Figure 13: The frequency step of algorithm *star*

umes. Indeed, tuple volumes capture less information than join tuples (note that the latter can produce the former but not the vice versa), and hence, are cheaper to calculate.

4.3 Conversion to Star-CNs

This section deals with the following *starization* problem: given a non-star CN, transform it to a star-CN CN' that returns the same set of MTJNTs, i.e., $MTJNT(CN) = MTJNT(CN')$. The goal is to minimize the total cost incurred in the transformation and executing the *star* algorithm (presented in Section 4.2) on CN'.

A basic observation is that, if CN has s vertices, then it has s equivalent star-CNs each of which has a different vertex as the root. Let us explain this with a concrete example. Consider the non-star CN in Figure 14a, corresponding to the schema graph in Figure 1 and a query keyword-set $Q = \{Tony, conf\}$. Figure 14b gives an equivalent star-CN CN_1', where $\texttt{WRITE}^{\emptyset}$ is the root. Notice that, conversion from CN to CN_1' requires a join between $\texttt{PAPER}^{\emptyset}$ and $\texttt{CONF}^{\{conf\}}$, and the result of the join becomes a leaf tuple-set in CN_1'. Similarly, Figure 14c shows another equivalent star-CN CN_2', which necessitates a join $\texttt{AUTHOR}^{\{Tony\}} \bowtie \texttt{WRITES}^{\emptyset}$. Figures 14d and 14e present the other two equivalent star-CNs CN_3' and CN_4'.

The quality of a star-CN CN' depends on two types of cost: the overhead of (i) converting CN to CN', and (ii) executing *star* on CN'. Hence, finding the optimal CN' would be trivial if we *were* able to predict both costs accurately. While the overhead of (ii) may be easy to estimate (as mentioned earlier, *star* scans each participating tuple-set twice, and writes the abridged root tuple-set once), predicting the overhead of (i) is hard for several reasons. First, join selectivity estimation is known to be a tricky problem [5]. Although there exist solutions [1] specifically designed for foreign-joins, they cannot be applied in our case, because the joins here – although they look like foreign-joins – are not exactly so. Recall that in a traditional foreign-join, every foreign key will *definitely* be joined with a primary key. This property no longer holds in our scenario due to the keyword-screening process. For example, consider the join $\texttt{AUTHOR}^{\{Tony\}} \bowtie \texttt{WRITES}^{\emptyset}$. Apparently, most foreign-key values in $\texttt{WRITES}^{\emptyset}$ will not find their matching primary-keys in $\texttt{AUTHOR}^{\{Tony\}}$, because $\texttt{AUTHOR}^{\{Tony\}}$ consists of only tuples containing the keyword *Tony*. Second, selectivity estimation demands specialized structures such as sample sets [5], synopses [1], etc. Such structures cannot be pre-computed because the tuple-sets of CN are dynamically generated according to the query keyword-set Q. Constructing the structures on the fly entails large cost itself [1, 5].

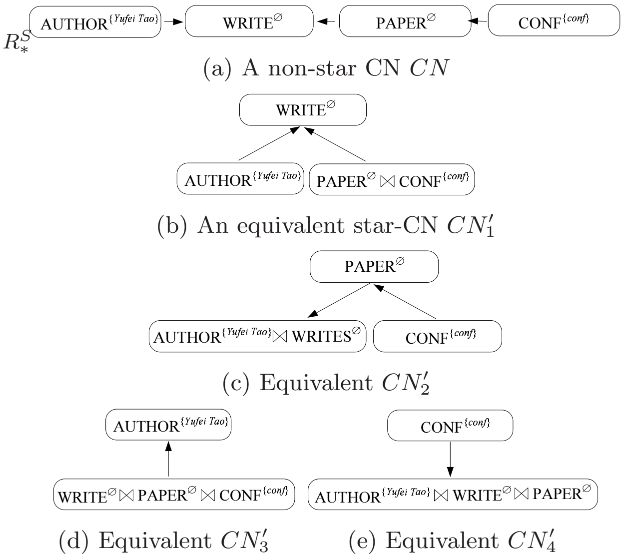


Figure 14: Multiple starization choices

A good strategy in starization is to avoid joins that produce gigantic results. For example, the CN_1' in Figure 14b is a poor choice, because $\texttt{PAPER}^{\emptyset} \bowtie \texttt{CONF}^{\{conf\}}$ essentially joins two sizable tuple-sets (in particular, $\texttt{CONF}^{\{conf\}}$ is the table $\texttt{CONF}$ itself – recall that every tuple in $\texttt{CONF}$ implicitly includes the table name as a term). Not only that the join itself incurs expensive cost, but also it creates a huge leaf tuple-set for CN_1', leading to large cost in the subsequent application of the *star* algorithm. The CN_2' in Figure 14c is a much better choice. In particular, $\texttt{AUTHOR}^{\{Tony\}}$ is a very small tuple-set. As a result, the join $\texttt{AUTHOR}^{\{Tony\}} \bowtie \texttt{WRITES}^{\emptyset}$ is fairly efficient, and produces only a small number of tuples.

Typically, a join is expensive if its participating tuple-sets have large cardinalities. There is a close connection between the size of a tuple-set and its type. We already know that a tuple-set R^S can be free or non-free. Here, we further divide non-free R^S into two types: R^S is *strongly non-free*, if S contains at least a keyword that is not the name of the raw table R; otherwise, R^S is *weakly non-free*. For example, $\texttt{AUTHOR}^{\{Tony\}}$ is a strongly non-free tuple-set, whereas $\texttt{CONF}^{\{conf\}}$ is weakly non-free. In general, weakly non-free and free tuple-sets are large, whereas a strongly non-free tuple-set is small[4], because usually only a fraction of the raw table R includes all the keywords in S. Hence, we should avoid joins that involve no strongly non-free tuple-set at all, e.g., $\texttt{PAPER}^{\emptyset} \bowtie \texttt{CONF}^{\{conf\}}$. These joins are said to be *bad*.

Motivated by this, we perform *starization* by choosing the star-CN that requires the least number of bad joins. In case there are multiple such star-CNs, we select the one with the greatest number of leaf tuple-sets (in general, the larger the number, the fewer pairwise joins are needed). To illustrate, consider the CN in Figure 14a. Among the equivalent star-CNs in Figures 14b-14e, CN_1' and CN_3' require one bad join, whereas CN_2' and CN_4' demand no bad join at all. Now we need to make a choice between CN_2' and CN_4'. As CN_2' has two leaves and CN_4' has only one, CN_2' requires fewer

[4]This observation was first made in [12].

Algorithm ***starization***

```
/* Input: a non-star CN with s tuple-sets R_1^{S_1}, ..., R_1^{S_s} */
initialize arrays bad-num and degree each with s elements
for i = 1 to s
    degree[i] = number of neighbors of R_i^{S_i} in CN
    remove R_i^{S_i} from the original CN, resulting in a set of
    connected components
    bad-num[i] = number of connected components that have
    at least two tuple-sets but no strongly non-free tuple-set
rt = ∅; min-bad-num = ∞; rt-degree = 0
for i = 1 to s
    if bad-num[i] < min-bad-num OR
    (bad-num[i] = min-bad-num AND degree[i] > rt-degree)
        rt = R_i^{S_i}
        min-bad-num = bad-num[i]; rt-degree = degree[i]
return the star-CN with root rt
```

Figure 15: The algorithm of *starization*

pairwise joins, and is the final output of *starization*.

It remains to clarify how to obtain the number of bad joins needed by a star-CN CN'. Assume that the root of CN' is R^S. Let us examine the vertex R^S in the original CN. Removal of R^S breaks CN into several connected components. The number of bad joins equals the number of components that have (i) at least two tuple-sets but (ii) no strongly non-free tuple-set. This number can be found with a single traversal of all the components.

Consider the CN in Figure 14a and $R^S = \texttt{WRITES}^{\emptyset}$. After deleting $\texttt{WRITES}^{\emptyset}$, the CN is partitioned into two components $\texttt{AUTHOR}^{\{Tony\}}$ and $\texttt{PAPER}^{\emptyset} \leftarrow \texttt{CONF}^{\{conf\}}$. The second component has two tuple-sets, neither of which is strongly non-free. Thus, we know that the star-CN rooted at $\texttt{WRITES}^{\emptyset}$ necessitates one bad join. Figure 15 formally summarizes the *starization* algorithm.

5. EXTENSIONS

Our analysis so far assumes that every occurrence of a term w is counted equally in its frequency, regardless of the raw table where w appears. Sometimes we may want to treat the occurrences in various tables differently. For example, a user, who wants to know more about the research of Tony, may consider terms in `PAPER` more important than those in `AUTHOR` (in the schema graph of Figure 1). For this purpose, s/he may give a higher weight to `PAPER` and a lower one to `AUTHOR`, so that every appearance of a term counts more in `PAPER` than `AUTHOR`.

Carrying the idea further, a more general method is to specify weights at the CN level. This is reasonable because a term from the same table may not necessarily have the same importance in different CNs. To explain, let us slightly modify the schema of Figure 1, by adding one more column *comments* to table `WRITES` (i.e., `WRITES` now has attributes *A_id*, *P_id*, *comments*). This new attribute records the comments of the author *A_id* on her/his paper *P_id*. Now consider a FCT query with keyword-set $Q = \{Tony, spatial, index\}$, and the following CNs:

$CN_1 : \texttt{AUTHOR}^{\{Tony\}} \rightarrow \texttt{WRITES}^{\emptyset} \leftarrow \texttt{PAPER}^{\{spatial,\ index\}}$

$CN_2 : \texttt{AUTHOR}^{\{Tony\}} \rightarrow \texttt{WRITES}^{\{index\}} \leftarrow \texttt{PAPER}^{\{spatial\}}$

Let w be a term in `PAPER`. Intuitively, an occurrence of w in (an MTJNT output by) CN_1 is more important than that in CN_2. This is because each MTJNT from CN_1 corresponds to a paper specifically on spatial indexing, whereas an MTJNT from CN_2 may be a paper on other spatial topics, but with a comment from Tony related to indexes.

Our FCT operator can be easily extended to incorporate weighting in the above scenarios. Actually, this is true both conceptually and algorithmically. In particular, conceptually, the only change necessary is the definition of function $count(T, w)$, which here returns the weighted number of occurrences of w in an MTJNT T. To elaborate the details, assume that CN is the candidate network that generates T. Suppose that CN has s tuple-sets $R_1^{S_1}$, ..., $R_s^{S_s}$, which, by the weighting rules in the underlying application, bear weights $wght_1$, ..., $wght_s$, respectively. Thus, $count(T, w)$ should be implemented as follows. First, we initialize a counter 0. Then, for every occurrence of w in T, we first obtain the tuple-set, say $R_i^{S_i}$, contributing the occurrence, and increase our counter by $wght_i$.

Accordingly, to support weighted FCT search, small changes are needed in the algorithms *starization* and *star* proposed in Section 4. Recall that, given a non-star candidate network CN, *starization* performs some preliminary joins to transform CN into a star-counterpart CN'. Each join produces a leaf tuple-set in CN'. To tackle a weighted FCT query, terms in the join result should be accompanied by the weights of their origin leaf tuple-sets. For example, let CN be as shown in Figure 14a. Converting it to the CN_2' in Figure 14c needs a join $\texttt{AUTHOR}^{\{Tony\}} \bowtie \texttt{WRITES}^{\emptyset}$. Then, for every occurrence of a term w in the join result, we associate it with the weight of $\texttt{AUTHOR}^{\{Tony\}}$ (or $\texttt{WRITES}^{\emptyset}$), if it comes from $\texttt{AUTHOR}^{\{Tony\}}$ (or $\texttt{WRITES}^{\emptyset}$).

Given a star-CN CN, on the other hand, *star* computes the total weighted occurrences *freq-CN*(CN, w) of each term w in the MTJNTs determined by CN. The only modification of *star* is in its frequency step, which computes *freq-CN*(CN, w) from the tuple volumes, by scanning each leaf tuple-set and the abridged root tuple-set once. Specifically, after fetching a tuple t, for every term w in t, we increase *freq-CN*(CN, w) by $vol(t) \cdot weight(t)$, where $vol(t)$ is the volume of t, and $weight(t)$ is the weight of the origin tuple-set of t.

Finally, it is worth mentioning that, since a FCT query concentrates on mining concepts, its effectiveness can be boosted when there is a *concept hierarchy*. This hierarchy captures the belonging-to relationships among terms; for instance, *nearest-neighbor* belongs to *spatial*. As a result, whenever *nearest-neighbor* is encountered in an MTJNT, we should increase the frequencies of both *nearest-neighbor* and *spatial*. This strategy makes it easier for FCT queries to discover related concepts at the high levels.

6. EXPERIMENTS

This section aims at achieving two objectives. First, in Section 6.1, we will demonstrate the usefulness of FCT search, i.e., it enables us to extract interesting information from real databases conveniently. Then, in Section 6.2, we will verify the efficiency of our FCT algorithm.

6.1 Effectiveness of FCT Search

We use a real database *IMDB* [17] that collects the cast, director, and genre information of over 800k movies and TV programs. Figure 16 presents the schema graph of *IMDB*, where the table names and columns are self-illustrative. The primary key of each table is underlined. Note that a movie may be classified in multiple genres, and thus, may have several records in `GENRE`. Furthermore, it is also possible that

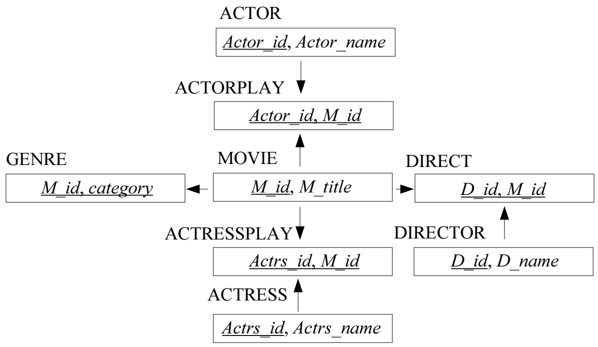


Figure 16: The schema graphs of *IMDB*

a movie does not belong to any genre, and hence, has no tuple in `GENRE`. The entire title of a movie, and the full name of an actor, actress, and director are treated as a single term. This is reasonable because a word appearing in, for example, the titles of two movies does not really bear any obvious meaning. Table 1 shows the cardinalities of the tables. Totally *IMDB* occupies 285 mega bytes.

To demonstrate the effectiveness of FCT retrieval, we will give the results of several representative queries, and explain why they are reasonable. Recall that a FCT query has two explicit parameters: a set Q of keywords, and the number k of terms requested. Furthermore, a FCT query also implicitly inherits another parameter from the traditional keyword search: R_{max}, which specifies the largest size of a CN, in terms of the number of participating tuple-sets. In the sequel, we fix k to 10 and R_{max} to 6.

First, we retrieve the most prolific comedy directors with

$Q1 = \{comedy,\ director\}$

yielding

Al Christie (365), Mack Sennett (303), Jules White (297) Friz Freleng (285), Allen Curtis (278), Chuck Jones (259) Dave Fleischer (255), Bud Fisher (252), William Beaudine (249), William Watson (219)

The number after each name corresponds to its frequency. Note that this result is obtained after removing the stopping words and the obvious noisy terms such as the table names `MOVIE` and `DIRECT`. All the directors in the result are highly successful directors in history. For example, Christie Al (1881-1951), a star on the Hollywood Walk of Frame, directed over 200 motion pictures, and is particularly well-known for his short comedies (*en.wikipedia.org/wiki/Al_Christie*).

A fan of Tom Hanks may be curious which director Tom is most mentioned with. This can be extracted by:

$Q2 = \{Tom\ Hanks,\ director\}$

with result

Louis Horvitz (9), Jeff Margolis (8), Dave Wilson (6) Beth Mccarthy Miller (5), Ron Howard (4), Laurent Bouzereau (4), David Frankel (4), Robert Zemeckis (3), David Leland (3), Penny Marshall (3)

Louis Horvitz, for instance, is indeed a director that has a close relationship with Tom. In 2002, Louis actually directed a TV program called *AFI Life Achievement Award: A Tribute to Tom Hanks*. To acquire the genres of the motion productions involving Tom Hanks, we perform

Table	Cardinality
ACTOR	741449
ACTORPLAY	4244600
ACTRESS	445020
ACTRESSPLAY	2262149
MOVIE	833512
DIRECTOR	121928
DIRECT	561173
GENRE	629195

Table 1: Table cardinalities of *IMDB*

$Q3 = \{Tom\ Hanks,\ genres\}$

returning

comedy (44), drama (34), short (20), family (17), romance (10), thriller (9), crime (8), music (8), fantasy (7), war (7)

Interestingly, while Tom Hanks is perhaps best known for his dramas, he actually took parts in many comedies as well (a recent one: *The Terminal*). Let us repeat the above two queries but with respect to Jim Carrey. Specifically,

$Q4 = \{Jim\ Carrie,\ director\}$

gives

Louis Horvitz (7), Bruce Gowers (4), Michel Gondry (3) Jeffrey Schwarz (3), Tom Shadyac (3), Bobby Farrelly (2) Peter Farrelly (2), Beth Mccarthy Miller (2), Troy Miller (2), Joel Schumacher (2)

The results of $Q2$ and Q_4 indicate that Louis Horvitz works closely with both Tom Hanks and Jim Carrie.

$Q5 = \{Jim\ Carrie,\ genres\}$

returns

comedy (40), short (14), drama (13), family (11), action (7), fantasy (7), music (7), adventure (6), crime (6), romance (6)

Jim Carrie is particularly mentioned only in one genre: comedy, whose frequency 40 is much higher than the others. As shown in $Q10$, Tom Hanks seems to be more versatile, by being heavily mentioned in both comedy and drama.

Finally, we show how to leverage FCT queries to discover the actors and actresses closely related to a director. For this purpose, we choose director Jules White (1900-1985), in the result of $Q1$, as a representative. The next query

$Q6 = \{Jules\ White,\ actor,\ comedy\}$

discovers

Moe Howard (108), Larry Fine (108), Vernon Dent (70) Shemp Howard (67), Al Thompson (49), Emil Sitka (46) Joe Palma (45), John Tyrrell (39), Johnny Kascier (37), Curly Howard (36)

The result is fairly reasonable. For example, Jules White is best known (*en.wikipedia.org/wiki/Jules_White*) for his short-subject comedies starring the "Three Stooges" – Moe Howard, Larry Fine, and Curly Howard – all of whom are included in the result. As for actresses, we run

$Q7 = \{Jules\ White,\ actress,\ comedy\}$

and obtain

	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	$Q7$
non-empty count	8	9	2	1	4	33	12

Table 2: CN statistics

Christine Mcintyre (38), Symona Boniface (23), Dorothy Appleby (21), Judy Malcolm (19), Nanette Bordeaux (15) Jean Willes (14), Barbara Jo Allen (14), Barbara Bartay (13), Margie Liszt (11), Harriette Tarler (11)

Again, these actresses are indeed highly relevant to Jules. For example, Christine Mcintyre stars, along with the Three Stooges mentioned earlier, in numerous 1950-comedies by Jules, including *Punchy Cow Punchers*, *Hugs and Mugs*, *Love at First Bite*, etc (*www.threestooges.com*).

Summary. The effectiveness of FCT search is reflected in two aspects. First, it is able to discover concepts that are highly related to the set of query keywords. Furthermore, the frequencies of those concepts generally indicate their importance. Second, a FCT query is easy to formulate. Specifically, as shown earlier, the keywords of all the queries $Q1$-$Q7$ are simple and intuitive. They can be provided even by non-database experts.

6.2 Efficiency of FCT Search

This section evaluates the efficiency of the proposed algorithm, referred to as *star-FCT*. As discussed in Section 4, *star-FCT* involves two components: (i) *star* (Figures 12 and 13), for aggregating term frequencies from a star-CN, and (ii) *starization* (Figure 15), for converting a non-star CN to a star counterpart. We compare our solution against the *baseline* approach. As mentioned in Section 4.1, given a FCT query with a keyword set Q and an integer k, *baseline* first solves a KS query with the same Q, computes the frequencies of all the terms in the result, and then, outputs the k most frequent terms.

All the results in the sequel are obtained on a computer running a Pentium IV dual-core CPU at 2.13GHz. To be fair for *baseline*, we minimize its cost by implementing a highly efficient join engine. In particular, our implementation incorporates the expression-sharing heuristic proposed in [12]. That is, after being computed, the result of a join is preserved, and re-used directly if the same join needs to be executed later. We allocate an equal amount of memory, 6 mega bytes, for both *star-FCT* and *baseline*. This memory buffer is significantly smaller than the size (over 280 mega bytes) of *IMDB*. It is worth noting that an efficient algorithm must be able to work with a small amount of memory because in practice the system may have to deal with numerous queries concurrently.

We will demonstrate the performance of the two algorithms on the queries $Q1$-$Q7$ analyzed in Section 6.1. Recall that, for each query, both *star-FCT* and *baseline* need to first enumerate all the CNs that have a chance to produce MTJNTs, in the way explained in Section 3. Many CNs are empty, i.e., they generate no MTJNTs at all. The number of non-empty CNs is an important indicator of the query overhead. In general, more non-empty CNs lead to higher query cost. Therefore, in Table 2, we list the number of non-empty CNs respectively for each query. Note that these numbers are identical for *star-FCT* and *baseline*.

Figure 17 presents the elapsed time of *star-FCT* and *baseline*. We break the performance of *star-FCT* into the overhead of *star* and *starization*, respectively. Above each column of *star-FCT*, we place a value denoting how much percent *starization* accounts for in the overall execution time. For example, for $Q1$, 36% of the *star-FCT* cost is due to *starization*. Evidently, *star-FCT* consistently outperforms *baseline*, achieving a maximum speedup ratio of 4 at $Q1$. As explained in Section 4.2, the superiority of *star-FCT* stems from the fact that it acquires the term frequencies without computing the join results of a star-CN at all.

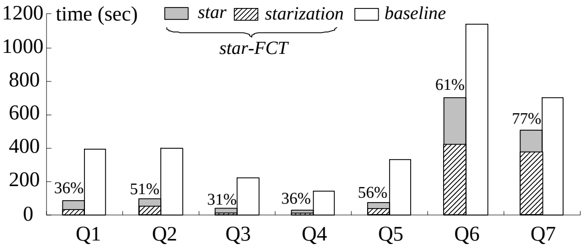


Figure 17: Efficiency comparison

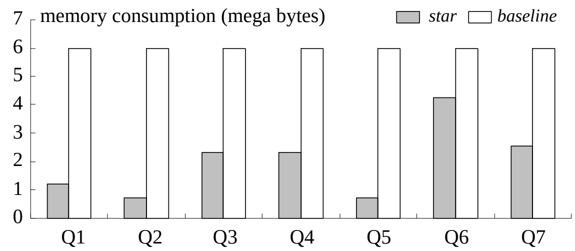


Figure 18: Memory consumption comparison

For most queries, *starization* entails only a fraction of the total cost of *star-FCT*, which confirms the effectiveness of our heuristics in Section 4.3. As expected, there is a strong correlation between the query cost (of both algorithms) and the number of non-empty CNs. For example, $Q6$ and $Q7$ are the two most expensive queries because they have the most non-empty CNs.

Our implementation of the *star* algorithm keeps the *num*- and *vol*-arrays memory resident (see Section 4.2). Next, we show that this is a reasonable choice, because these arrays are so small that they easily fit in the memory. For this purpose, we measure the largest memory consumption of *star* during its execution, and compare it with *baseline*. As mentioned earlier, the limit of memory usage is 6 mega bytes for each algorithm.

Figures 18 presents the results. *Baseline* always uses up all the available memory, because its join engine automatically makes full use of memory to reduce the join overhead. The consumption of *star*, on the other hand, varies across queries. This is not surprising because different queries demand *num*- and *vol*-arrays with different sizes, depending on the characteristics of the CNs generated. In all queries, *star* requires no more than 5 mega bytes of memory. Even in the worst case ($Q5$), *star* takes up less than 6 mega bytes. Finally, note that the above results apply to *star*. As with *baseline*, the other component *starization* of *star-FCT* also utilizes all the vacant memory to minimize the cost of joins.

Summary. The proposed *star-FCT* algorithm is able to solve FCT queries efficiently. In most cases, the cost of *star-FCT* is significantly dominated by its *star* component, thus justifying the sophisticated heuristics in *star*. Furthermore, *star-FCT* requires a small amount of memory, even when the underlying database is larger than 280 mega bytes.

7. CONCLUSIONS

This paper proposes a novel operator called frequent co-occurring term (FCT) search. Given a set Q of keywords and an integer k, a FCT query returns the k terms that appear most frequently in the result of a traditional KS (keyword search) query. Unlike KS that produces joined tuples containing all the keywords in Q, FCT search aims at extracting the terms that most accurately characterize Q. We devise a new algorithm that efficiently solves a FCT query without resorting to conventional KS methods. As experimentally evaluated with real data, (i) FCT search can indeed discover highly intuitive observations that cannot be found via ordinary KS queries; (ii) our FCT algorithm is fairly efficient, and requires small memory space.

Our study also points to several promising topics for future research. As shown in Figure 18, the *star* algorithm typically does not consume all the memory available. Thus, an interesting direction is to investigate the possibility of utilizing the remaining memory to further lower the execution cost. Furthermore, we have considered only static data. Maintenance of FCT results over a continuous data stream demands alternative strategies to be explored. Finally, our discussion has focused exclusively on relational databases. Extending FCT queries to other types of data, such as XML documents and spatial entities, deserves careful consideration.

Acknowledgements

This work was partially supported by CERG grants CUHK 1202/06, 4161/07, 4173/08, and 4182/06 from HKRGC.

REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. of ACM Management of Data (SIGMOD)*, pages 275–286, 1999.

[2] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *Proc. of ACM Management of Data (SIGMOD)*, pages 575–586, 2006.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 564–575, 2004.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[5] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. of ACM Management of Data (SIGMOD)*, pages 263–274, 1999.

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

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[8] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proc. of International Conference on Data Engineering (ICDE)*, 2008.

[9] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *Proc. of ACM Management of Data (SIGMOD)*, pages 16–27, 2003.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *Proc. of ACM Management of Data (SIGMOD)*, pages 305–316, 2007.

[11] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 850 – 861, 2003.

[12] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 670–681, 2002.

[13] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.

[14] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 505–516, 2005.

[15] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 173 – 182, 2006.

[16] F. Lui, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 563–574, 2006.

[17] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k keyword query in relational databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 115–126, 2007.

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

[19] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 346–355, 2007.

[20] P. Wu, Y. Sismanis, and B. Reinwald. Towards keyword-driven analytical processing. In *Proc. of ACM Management of Data (SIGMOD)*, pages 617–628, 2007.

[21] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 527–538, 2005.

[22] B. Yu, G. Li, K. Sollins, and A. Tung. Effective keyword-based selection of relational databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 139–150, 2007.