Universal Top-k Keyword Search over Relational Databases

Ning Zhang
n25zhang@uwaterloo.ca

Ihab F. Ilyas
ilyas@uwaterloo.ca

M. Tamer Özsu
tozsu@uwaterloo.ca

University of Waterloo
David R. Cheriton School of Computer Science
Waterloo, Canada

Technical Report CS-2011-03
January, 2011

ABSTRACT

Keyword search is one of the most effective paradigms for information discovery. One of the key advantages of keyword search querying is its simplicity. There is an increasing need for allowing ordinary users to issue keyword queries without any knowledge of the database schema. The retrieval unit of keyword search queries over relational databases is different than in IR systems. While the retrieval unit in those IR systems is a document, in our case, the result is a synthesized document formed by joining a number of tuples.

We measure result quality using two metrics: *structural quality* and *content quality*. The *content quality* of a JTT is an IR-style score that indicates how well the information nodes match the keywords, while the *structural quality* of JTT is a score that evaluates the meaningfulness/semantics of connecting information nodes, for example, the closeness of the corresponding relationship. We design a hybrid approach and develop a buffer system that dynamically maintains a partial data graph in memory. To reuse intermediate results of SQL queries, we break complex SQL queries into two types of simple queries. This allow us to support very large databases and reduce redundant computation. In addition, we conduct extensive experiments on large-scale real datasets to study the performance of the proposed approaches. Experiments show that our approach is better than previous approaches, especially in terms of result quality.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

General Terms

Algorithms, Experimentation, Performance

Keywords

top-k, keyword search, relational databases, information retrieval, ranking

1. INTRODUCTION

Commercial relational database management systems (DBMS) generally provide querying capabilities for text attributes. However, this search functionality requires queries to be specified on an exact database schema. However, under this model, (1) users need to know the schema; (2) users need knowledge about full-text search functionality provided by the particular system; and (3) users need to form a complex structured query to synthesize meaningful results (records) that contain required information. In many applications, these restrictions significantly hinder usability.

For example, consider the personal information database given in Table 1. One might want to search a person named *'Joan'* who has a niece named *'Caroline'*. However, the 'aunt-niece' relationship is not a direct relationship in the database. We can synthesize 'aunt-niece' relationship by a corresponding four-way join SQL query shown in Figure 1. To compose such a SQL query, users need to know the schema of table *Person* and the *CONTAINS* clause for full-text search. Moreover, there are four joins involved in

(a) Person

rid	identity_number	person_name	place_of_birth	farther	mother	spouse	Alma_mater
h_1	ID00000001	Joseph P. Kennedy	p_1	NULL	NULL	h_2	e_1
h_2	ID00000002	Rose Fitzgerald Kennedy	p_1	NULL	NULL	h_1	NULL
h_3	ID00000003	John F. Kennedy	p_2	h_1	h_2	h_4	e_2
h_4	ID00000004	Jacqueline Kennedy Onassis	p_3	NULL	NULL	h_3	e_3
h_5	ID00000005	Caroline Kennedy	p_4	h_3	h_4	NULL	e_2
h_6	ID00000006	Edward M. Kennedy	p_1	h_1	h_2	h_7	e_1
h_7	ID00000007	Joan Bennett Kennedy	p_4	NULL	NULL	h_6	e_4

(b) Place

rid	place_name
p_1	Boston, Massachusetts
p_2	Brookline, Massachusetts
p_3	Southampton, New York
p_4	New York City, New York

(c) Institution

rid	institution_name
e_1	Harvard College
e_2	Harvard University
e_3	George Washington University
e_4	Manhattanville College

Table 1: a hypothetic personal information database

JTT-1: $h_7 \xrightarrow{spouse-spouse} h_6 \xrightarrow{child-father} h_1 \xleftarrow{father-child} h_3 \xleftarrow{father-child} h_5$

JTT-2: $h_7 \xrightarrow{place\ of\ birth} p_4 \xleftarrow{place\ of\ birth} h_5$

Figure 2: Possible results

```
SELECT  *
FROM    Person H1, Person H2, Person H3,
        Person H4, Person H5
WHERE   H1.spouse = H2.rid
AND     H2.father = H3.rid
AND     H4.father = H3.rid
AND     H5.father = H4.rid
AND     CONTAINS(H1.name, 'Joan', 1) > 0
AND     CONTAINS(H5.name, 'Caroline', 2) > 0
```

Figure 1: An Oracle SQL Example

this complex SQL query. Keyword search over relational databases has become an important alternative to tackle this problem.

1.1 Background

One way to execute a keyword search query would be to list all records (tuples or multiple tuple joined with some join condition) that contain the given keywords. These records are ranked according to a scoring function that reflects matches between keywords and results. A well-designed scoring function will report more meaningful results first. For example, the 'aunt-niece' relationship between *Caroline* and *Joan* might rank in the top-k answers in the output of a keyword search query on *'Joan'* and *'Caroline'*.

Keyword search has been extensively studied within the context of information retrieval (IR) systems. However, the retrieval unit of keyword search queries in DBMSs is different than that in IR systems. While the retrieval unit in IR systems is a single (real) document, in our case, the result is a logical (virtual) database record formed by joining a number of tuples. This is commonly referred to as a *joining tree of tuples* (JTT) [6, 5] or a *joined tuple tree* [10]. Consider a relational database schema $\mathcal{R} = \{R_1, \ldots, R_n\}$. The *schema graph* (SG) is a directed graph that captures relationships between relations. SG has a node s_i for each relation R_i. If there is a foreign key/primary key relationship from relation R_i to relation R_j, then there is an edge from node s_i to node s_j in SG. In addition, we define the *data graph* (DG) as follows. For each tuple t in the database, DG has

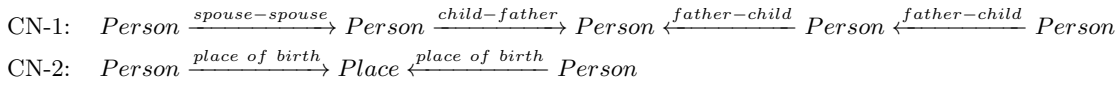


Figure 3: Corresponding CNs

a corresponding node d_t. If t_i has a foreign key referencing t_j, DG contains an edge from d_{t_i} to d_{t_j}. For each node d_t in DG, there is a corresponding node in SG. For each edge $d_{t_i} \rightarrow d_{t_j}$ in DG, if $t_i \in R_i$ and $t_j \in R_j$, there is a corresponding edge $s_i \rightarrow s_j$ in SG because $t_i \bowtie t_j \in R_i \bowtie R_j$. A JTT is a connected acyclic subgraph (tree) of DG.

Some tuples in a JTT are information nodes that match the given keywords while others help to connect the information nodes. For example, in JTT-1 (Figure 2), tuple h_7 (matching keyword *'Joan'*) and tuple h_5 (matching keyword *'Caroline'*) are information nodes, while h_6, h_1 and h_3 help to connect them. All leaf nodes in a JTT are information nodes.

The quality of a JTT depends mainly on two components: the *content quality* and the *structural quality*. The *content quality* of a JTT is an IR-style score that indicates how well the information nodes match the keywords, while the *structural quality* of JTT is a score that evaluates the meaningfulness/semantics of connecting information nodes, for example, the closeness of the corresponding relationship.

Consider the query above involving two keywords: w_1 = *'Joan'* and w_2 = *'Caroline'*. This query result might contain a single information node or two information nodes, one matching w_1 and the other matching w_2, and there is a close relationship between these two nodes. In the example given in Table 1, there is no single tuple corresponding to both keywords; however, there are two tuples: $\langle$ h_7, Joan Bennett Kennedy $\rangle$ corresponding to w_1 and $\langle$ h_5, Caroline Kennedy $\rangle$ corresponding to w_2. A JTT can be generated in two ways that join h_7 and h_5 to obtain the result; these are shown in Figure 2.

Existing solutions to keyword search over relational databases can be roughly classified into two categories: (1) *the implicit data graph model* and (2) *the explicit data graph model*. For example, DBXplorer [1], DISCOVER [6], DISCOVER2 [5], SPARK [10], proposal in Liu et al. [9] belong to the first class and BANKS [2], Kacholia et al. technique [8], and BLINKS [4] belong to the second class.

Explicit data graph approaches materialize DG and keep the entire DG in memory. These approaches can directly access the tuple nodes during the search procedure. Starting from nodes containing query keywords, they heuristically expand the result from a single node to a Steiner tree [7] containing all given keywords. Implicit data graph approaches, however, leave the DG in the underlying DBMS and only keep the SG in memory. Relevant parts of the graph are accessed through invocation of multiple SQL queries.

1.2 Motivation

Most implicit data graph approaches measure the *structural quality* of a JTT by its size, defined as the number of tuples in a JTT. For example, in Figure 2, the size of JTT-1 is five while the size of JTT-2 is three. Existing implicit data graph approaches will rank JTT-2 higher than JTT-1 (because the size of JTT-2 is smaller). However, JTT-1 shows that *Caroline Kennedy* is a niece of *Joan Bennett Kennedy* while JTT-2 shows that these two people are born in the same city $\langle p_4, \text{New York City, New York} \rangle$. One can argue that the aunt-niece relationship is much stronger than the fellow-townsman relationship; therefore, JTT-1 should be ranked higher than JTT-2.

In contrast, explicit data graph approaches capture the importance of the JTT by measuring *structural quality* of JTT as the sum of edge weights (reflecting the coherence of the relationship between two tuples) in a JTT. On the other hand, current implementations of explicit data graph approaches have several restrictions in measuring *content quality*. For example, these approaches require that the *content quality* of JTT must be computed in a distributive manner: IR-style scores for individual tuples in a JTT are calculated separately and then summed up. With this restriction, they can use simple heuristics in a greedy fashion to search a Steiner [7] tree connecting all keywords. This restriction improves efficiency but the *content quality* is compromised. For example, in Figure 2, JTT-1 consists of five tuples, each corresponding to a person in the Kennedy fam-

ily. Suppose the query involves three keywords: w_1 = *'Joan'*, w_2 = *'Caroline'*, and w_3 = *'Kennedy'*. All five tuples in JTT-1 match w_3 once. The total score of all five occurrences of w_3 is five times the score of a single occurrence. The score for keyword w_3 then dominates the final IR score. Implicit data graph approaches do not impose this restriction and the combining function can be more sophisticated than simple summation.

In terms of run-time efficiency, explicit data graph approaches are better because they can access data directly while implicit data graph approaches can only access data indirectly through SQL queries, which may incur high I/O cost. In terms of storage efficiency, implicit data graph approaches are better because they keep only SG in memory while explicit data graph approaches keep the entire DG in memory.

Considering the mentioned tradeoffs, there is no clear winner between explicit data graph and implicit data graph approaches. Hence, we design a hybrid approach that incorporates a better *structural quality* measure similar to explicit data graph approaches, and a better *content quality* measure similar to implicit data graph approaches. We also develop a buffer system that dynamically maintains a partial DG in memory for better execution performance.

1.3 Outline of the Paper

In Section 2, we introduce the problem definition. In Section 3, we present an overview of our solution. In Section 4, we study the *structural quality* of JTT. In Section 5, we study the *content quality* of JTT. In Section 6, we introduce our execution engine, including a non-redundant CN generator, a revised version of Block Pipeline Algorithm, a connection test algorithm, and a buffer system. In Section 7, we present experimental results to demonstrate the effectiveness and efficiency of our approach. In Section 8, we summarize conclusions of our work.

2. PROBLEM DEFINITION

We begin with the database schema, schema graph (SG), and data graph (DG) definitions given earlier. The retrieval unit is a JTT. We formally define a top-k keyword search query as follows:

DEFINITION 1. *A query consists of a set of keywords, $Q = \{w_1, w_2, \ldots, w_{|Q|}\}$, and a parameter k indicating that a user is only interested in top-k results ranked by scores associated with each result.*

We define the quality of a JTT as follows:

DEFINITION 2. *Given a query Q and a JTT T, the result quality of T is measured by a ranking score*

$$score(T, Q) = f_{structural}(T) \times f_{content}(T, Q)$$

, where $f_{structural}(T)$ measures the structural quality of T and $f_{content}(T, Q)$ measures the content quality of T. Higher scores mean better quality.

We will formally define $f_{structure}(T)$ and $f_{content}(T, Q)$ in Sections 2 and 3, respectively.

DEFINITION 3. *For each node d_t in a JTT T, there is a corresponding node in SG. For each edge $d_{t_i} \rightarrow d_{t_j}$ in T, if $t_i \in R_i$ and $t_j \in R_j$, there is a corresponding edge $s_i \rightarrow s_j$ in SG. Each JTT corresponds to a connected non-simple subgraph of SG, which is referred to as the Candidate Network (CN).*

While JTTs are instance subgraphs in DG, CNs are schema subgraphs in SG. Note that the same relation can appear multiple times in a CN. For example in Table 1, two JTTs, $h_1 \xleftarrow{father-child} h_3$ and $h_3 \xleftarrow{father-child} h_5$ both correspond to CN: $Person \xleftarrow{father-child} Person$, which represents the father-child relationship. Relation $Person$ appears twice in the CN.

DEFINITION 4. *The relations in the CN are also called **tuple sets**. There are two kinds of tuple sets: those tuples matching at least one keyword are called **non-free tuple sets** $R^Q = \{t | t \in R, t$ contains at least one keyword $w_i \in Q\}$, and others are called **free tuple sets** $R^{\{\}} = R$. A **non-free tuple set** in a CN corresponds to an information node in a JTT.*

Since all leaf nodes of a JTT are information nodes, all leaf nodes of a CN are *non-free tuple sets*.

Finally, we formally define our problem as follows:

DEFINITION 5. *Given a query $Q = \{w_1, w_2, \ldots, w_{|Q|}\}$ and a parameter k, the answer of a top-k query is a list of the k JTTs, $T_1, \ldots, T_k$, whose ranking scores ($score(T_i, Q)$) are the highest.*

3. OVERVIEW OF OUR SOLUTION

In this paper, we first define *structural quality* of JTT as the *coherence score* of the corresponding CN (Section 2). We assign all JTTs corresponding to a CN the same *structural quality* score since the *structural quality* depends on the semantics of the CN that is shared among all JTTs.

We propose a novel IR-style score as *content quality* of JTT (Section 3). Both Page-rank [3] style score and TF-IDF style score are incorporated. We also adopt the idea of the use of a single tuning parameter to inject AND or OR semantics into the ranking formula [10], although we define the tuning parameter in a different way. Our IR-style scoring function is calculated for the whole JTT without any assumptions, such as distributive property or assuming that tuples matching a certain keyword are uniformly and independently distributed in each relation.

In implicit data graph approaches, most intermediate results of SQL queries overlap, resulting in repeatedly computing the same intermediate result by different SQL queries. We develop a buffer system that dynamically maintains a partial data graph in memory and we only fetch data through SQL queries when it is necessary (Section 4). In this way, we can support very large databases and reduce redundant computation. Our approach is more efficient than implicit data graph model approaches and does not suffer from an excessive I/O cost when the data size exceeds the memory size.

4. PROPOSED STRUCTURAL QUALITY METRIC

There is a common intuition behind all existing *structural quality* definitions. They all try to define a "structural size" of a JTT and then use the inverse of this "structural size" to measure the *structural quality*. Many "structural size" definitions have been proposed. For example, existing implicit data graph approaches simply use the total number of tuples in a JTT as the "structural size" and existing explicit data graph approaches use the aggregate of edge weights as the "structural size".

There is a common underlying assumption: it is easier to find a JTT matching all given keywords in a "larger" CN. The "structural size" reflects how easily we can find a JTT that matches given keywords in the given CN. So, the "structural size" is an approximate measure of the probability of the existence of a JTT matching given keywords in the given CN. That is why all existing approaches penalize those answers with large "structural size". Instead of defining a new "structural size", we deal with the probability of the existence of a JTT matching randomly picked keywords in the given CN directly.

4.1 Coherence Score

As mentioned earlier, we assign all JTTs corresponding to the same CN with the same *structural quality* score since the *structural quality* depends on the semantics of the CN that is shared among all corresponding JTTs. So, we define the *structural quality* of a JTT T ($f_{structural}(T)$) as the *coherence score* of the corresponding CN, defined as $C(CN) = f(size(CN), NCS(CN))$, where $size(CN)$ is the total number of tuple sets in CN and $NCS(CN)$ is the *normalized connectivity score*, which is defined as the normalized score that reflects the probability of existence of a JTT matching randomly picked keywords. A higher *normalized connectivity score* implies a stronger connection among tuples in a CN. We will discuss the formal definition of $NCS(CN)$ later.

We consider the range of *coherence score*s. The *coherence score* for the closest relationship should be 1 because there is no penalty for the *structural quality*. The *coherence score* for the loosest relationship should be $1/size(CN)$.

DEFINITION 6. *The coherence score of a CN is defined as follow:*

$$C(CN) = \max\{1, \frac{1}{size(CN) \cdot (1 - NCS(CN))}\}$$

4.2 Connectivity Score

Before introduce the definition of *normalized connectivity score*, we first introduce *connectivity score*. Recall that there are two kinds of tuple sets: free tuple sets and non-free tuple sets. Those tuples in the non-free tuple set match at least one keyword. We define a random process that picks non-free tuples at random from the non-free tuple sets of the CN. The *connectivity score* is estimated by the probability that the randomly picked non-free tuples are connected with some free tuples, or, $CS(CN) = Pr\{\exists$a set of free tuples that

connects randomly picked non-free tuples}. More formally, we define $CS(CN)$ as follows.

DEFINITION 7. *Given a CN, which contains a set of non-free tuple sets $U_i = \{u_{i,1}, \ldots, u_{i,|U_i|}\}, 1 \leq i \leq n$ and a set of free tuple sets $V_i = \{v_{i,1}, \ldots, v_{i,|V_i|}\}, 1 \leq i \leq m$, the connectivity score, $CS(CN)$ is defined as the probability that the randomly picked non-free tuples $u'_1 \in U_1, \ldots, u'_n \in U_n$ are connected with some free tuples $v'_1 \in V_1, \ldots, v'_m \in V_m$.*

For example in Table 1, there are two CNs — CN_1: $Person \rightarrow Institution \leftarrow Person$ (alumni), and CN_2: $Person \xrightarrow{child-father} Person \xleftarrow{father-child} Person$ (siblings). Then, $CS(CN_1) = Pr\{\exists$a tuple $p_0 \in Institution$ that connects $p_1 \in Person$, $p_2 \in Person$ while p_1, p_2 are randomly picked from $Person\}$. If we randomly choose two people, the probability that two people go to the same school is much higher than the probability that two people have the same father. The alumni relationship is more common than the sibling relationship; therefore CN_2 has a higher coherence score than CN_1. As a rule, a CN with a lower *connectivity score* is assigned a higher *coherence score* for *structural quality*.

To calculate this *connectivity score*, we assume uniformity and independency. For any two adjacent relations P, Q in SG, $P \rightarrow Q$, if we randomly choose two tuples $p \in P$, $q \in Q$, the probability that p, q are connected ($Pr\{p \rightarrow q\}$) is $1/|Q|$ (uniformity). For any three directly connected relations in SG, $P \leftrightarrow Q \leftrightarrow R$, if we randomly choose three tuples $p \in P$, $q \in Q$, and $r \in R$, the probability that p, q, r are connected is $Pr\{p \leftrightarrow q\} \times Pr\{q \leftrightarrow r\}$ (independence).

We propose a tree structure to recursively compute the *connectivity score*. There is a node for each tuple set in the CN and there is an edge from node i to node j if and only if the corresponding tuple set i and j are directly connected in the CN. We can choose any node as the tree root. Our computation starts from the root and moves down to the leaves. Details of the tree construction and recursive computations are in Appendix.

For example in Table 1 (statistics are shown in Table 2), $CS(Person \rightarrow Institution \leftarrow Person) = 1/|Institution| = 0.002$, $CS(Person \xrightarrow{child-father} Person \xleftarrow{father-child} Person) = 1/|Person| = 0.000001$. Based on our calculation, the sibling relationship should get a higher *coherence score* for the *structural quality* because it has a much lower *connection probability*. If we can actually find a JTT corresponding to the sibling relationship, which matches all given keywords, it is very likely to be the answer because such a low probability event implies that it is very likely that the user knows this exact relationship.

Relations	# of tuples
Person	1000000
Place	50
Institution	500

Table 2: hypothetic statistics of the Personal Information database

4.3 Normalized Connectivity Score

The *connectivity score* is a measure of connection strength among a set of non-free tuple sets. Suppose there are n non-free tuple sets, $U_1, U_2, \ldots, U_n$. We setup an ideal *connectivity score* for these tuple sets. Based on these n non-free tuple sets, we construct a hypothetical CN by adding a hypothetical connection node V that contains only one tuple v. In addition, V has a foreign key referring to each non-free tuple set $U_i, 1 \leq i \leq n$. The *connectivity score* of this hypothetical CN is $1/(|U_1| \times \cdots \times |U_n|)$. Since this hypothetical CN has the most compact structure to connect all given tuple sets, we take its *connectivity score* as the ideal *connectivity score*. We can normalize the *connectivity score* of a CN as follow:

DEFINITION 8. *Given a CN, which contains a set of non-free tuple sets $U_1, U_2, \ldots, U_n$, the normalized connectivity score is defined as*

$$NCS(CN) = \frac{\log(CS(CN))}{\log(1/(|U_1| \times \cdots \times |U_n|))}$$

For a given JTT, we use the *coherence score* of its corresponding CN to measure the *structural quality*. So, in our formulation, *structural quality* is defined as:

DEFINITION 9.

$$f_{structure}(T) = \max\{1, \frac{1}{size(CN) \cdot (1 - NCS(CN))}\}$$

.

For example in Figure 3, CN_1 (aunt-niece relationship) is the CN corresponding to JTT-1 and CN_2 (fellow townsman

relationship) is the CN corresponding to JTT-2. Using our definition of *coherence score*, we have the following results: $C(CN_1) = 0.4$ and $C(CN_2) = 0.38$. Hence, we rank JTT-1 higher than JTT-2 because the CN corresponding to JTT-1 has a larger *coherence score*, which implies a stronger relationship. As mentioned earlier, existing implicit data graph approaches would rank JTT-2 higher than JTT-1 because the number of tuples in JTT-2 is less than the number of tuples in JTT-1. Our result is more reasonable because the aunt-niece relationship is stronger than the fellow townsman relationship.

5. PROPOSED CONTENT QUALITY METRIC

Existing approaches to computing *content quality* of a JTT use two basic IR-style scores: Page-Rank [3] and TF-IDF. The Page-Rank score emphasizes the importance of a tuple as a whole, while the TF-IDF score emphasizes keyword matching.

There are two existing methods to calculate the global TF-IDF score of a JTT: (1) the local TF-IDF style scores for individual tuples in a JTT are calculated separately and then combined as a global score (Figure 4); or (2) a JTT is treated as a virtual document by concatenating the text contents of tuples in the JTT and a TF-IDF score for the virtual document is calculated by assuming tuples matching a certain keyword are uniformly and independently distributed in each relation (Figure 5). Both methods have drawbacks. Consider a query involving two keywords and a JTT assembled by joining two tuples, and assume that each tuple matches only one keyword. The first method does not distinguish between matching one keyword twice (both tuples match the same keyword) and matching two keywords once (two tuples match different keywords). The assumptions about uniformity and independence are too strong for the second method to work. For example in Table 1, 'John' is a person's name and 'University' usually appears in the name of an educational institution. Such keywords are not uniformly and independently distributed in the *Person* and the *Institution* relations.

We propose a novel IR-style score as *content quality* of a JTT. Both Page-Rank style score and TF-IDF style score

$$\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \xrightarrow{TF-IDF} \begin{pmatrix} tfidf(t_1,w_1) & \dots & tfidf(t_1,w_m) \\ \vdots & \ddots & \vdots \\ tfidf(t_n,w_1) & \dots & tfidf(t_n,w_m) \end{pmatrix}$$

$$\downarrow \sum$$

$$globalscore \xleftarrow{combine} \begin{pmatrix} \sum tfidf(t_1,w_j) \\ \vdots \\ \sum tfidf(t_n,w_j) \end{pmatrix}$$

Figure 4: method (1)

$$\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \xrightarrow{concatenate} \sum t_i$$

$$\downarrow \textit{TF-IDF}$$

$$globalscore \xleftarrow{combine} \begin{pmatrix} tfidf(\sum t_i,w_1) \\ \vdots \\ tfidf(\sum t_i,w_m) \end{pmatrix}$$

Figure 5: method (2)

$$\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \xrightarrow{TF-IDF} \begin{pmatrix} tfidf(t_1,w_1) & \dots & tfidf(t_1,w_m) \\ \vdots & \ddots & \vdots \\ tfidf(t_n,w_1) & \dots & tfidf(t_n,w_m) \end{pmatrix}$$

$$\downarrow \sum$$

$$globalscore \xleftarrow{combine} \begin{pmatrix} \sum tfidf(t_i,w_1) & \dots \sum tfidf(t_i,w_m) \end{pmatrix}$$

Figure 6: our method

are incorporated in our scoring function without assuming uniformity and independence.

Note that both methods described above obtain a *local score* ($\sum tfidf(t_i, w_1)$ and $tfidf(\sum t_i, w_1)$) for each tuple before combining them as one *global score*. However, our method does not obtain a numeric local score for each tuple. Instead, we obtain a local score vector for each tuple to calculate a numeric score for each keyword (Figure 6).

5.1 Local Score

We adopt the following basic TF-IDF ranking formula:

$$tfidf(t, w_i) = \begin{cases} 0 & tf_{w_i}(t) = 0 \\ \frac{1+\log(1+\log(tf_{w_i}(t)))}{(1-s)+s\frac{t.dl}{avdl}} \cdot \log(idf_{w_i}) & tf_{w_i}(t) > 0 \end{cases}$$

The above function is the TF-IDF score of a single tuple t according to keyword w_i. If we concatenate all text attributes first (method 2), the term frequencies and inverse document frequencies of an individual text attribute are lost. Then, we have to estimate global term frequencies and inverse document frequencies instead of using original individual ones. As we mentioned earlier, keywords are not uniformly and independently distributed in different relations. Since estimation error affects the global *content quality* score, it is better not to concatenate text attributes. After applying the TF-IDF method, we can get a matrix of TF-IDF terms. We have two choices — either combine terms by row first or by column first. Existing approaches combine TF-IDF terms by row first (method 1). As we mentioned earlier, method 1 does not distinguish between matching one keyword twice (both tuples match the same keyword) and matching two keywords once (two tuples match different keywords).

To overcome the drawbacks previously discussed, we can compute the content score of a single tuple t as a vector $(tfidf(t, w_1), \ldots, tfidf(t, w_m))$. Our *local score* is an m-dimensional vector. In addition, Page-Rank score $PG(t)$, which is query-agnostic, is also used in our *local score*.

DEFINITION 10. *The score vector of a tuple t (local score) is defined as:*

$$\begin{aligned} \overrightarrow{text(t, Q)} &= PG(t)\big(tfidf(t, w_1), \ldots, tfidf(t, w_m)\big) \\ &= \Big(PG(t)tfidf(t, w_1), \ldots, PG(t)tfidf(t, w_m)\Big) \end{aligned}$$

In this way, all TF-IDF information of an individual tuple is preserved. However, comparing to numeric *local score*, the trade-off is the complex vertex-form representation. Then, we will define our *global score* based on *local score* vectors.

5.2 Global Score

Suppose there are l non-free tuples, $t_1, \ldots, t_l$, in JTT T. We have the following *local scores*: $\overrightarrow{text(t_1, Q)}, \ldots, \overrightarrow{text(t_l, Q)}$. As we discussed, we sum up TF-IDF term by column first.

DEFINITION 11. *The global score vector of a JTT T is defined as:*

$$\begin{aligned} &\overrightarrow{text(T, Q)} = \Big(\overrightarrow{text(t_1, Q)}, \ldots, \overrightarrow{text(t_l, Q)}\Big) \\ &= \Big(\textstyle\sum_{i=1}^{l} PG(t_i)tfidf(t_i, w_1), \ldots, \sum_{i=1}^{l} PG(t_i)tfidf(t_i, w_m)\Big) \end{aligned}$$

In this way, we group term frequency counts by corresponding keyword. In contrast, existing methods group term frequency counts by corresponding tuple. For *content quality*, we believe choosing corresponding keywords as primary dimensions is more reasonable. After we obtain a global score vector $\overrightarrow{text(T, Q)}$ with dimensions corresponding to given keywords, we need to calculate a numeric *global score* $text(T, Q)$ from the score vector.

DEFINITION 12. *The **global score** of a JTT T is defined as:*

$$\begin{aligned} text(T, Q) &= \Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} PG(t_i)tfidf(t_i, w_j)\big)^{\frac{1}{p}}}{m}\Big)^p \\ &= \Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} PG(t_i)tfidf(t_i, w_j)\big)^{\frac{1}{p}}}{m}\Big)^p \end{aligned}$$

where p is a tuning parameter [10].

Note that p can smoothly switch the completeness factor biased towards the OR semantics to the AND semantics, when p increases from 1.0 to ∞. If p is set to 1.0, matching one keyword twice and matching two keywords once will be treated equally. If p is set to ∞, it is not allowed to miss any keyword in the result. Usually, a p value of 3.0 is sufficient for almost all cases. Therefore, we use the *global score* to measure the *content quality*. So, in our formulation, $f_{content}(T, Q) = text(T, Q)$.

To apply top-k accelerating algorithms such as skyline algorithms in the following sections, we need to provide a monotonic upper bound for our *global score*.

For simplicity, we denote $PG(t_i)tfidf(t_i, w_j)$ by $x_{i,j}$, where

$PG(t_i)$ is a term irrelevant to the query. Then,

$$text(T,Q) = \Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} x_{i,j}\big)^{\frac{1}{p}}}{m}\Big)^{p}.$$

Let $LS_i = \sum_{j=1}^{m} x_{i,j}$. This can be computed locally for each tuple t_i.

THEOREM 1.

$$\Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} x_{i,j}\big)^{\frac{1}{p}}}{m}\Big)^{p} \le \frac{m'^{p-1}}{m^p}\frac{\sum_{i=1}^{l} S_i}{m}, p \ge 1$$

, where m' is the number of matched keywords.

Proof: Let $y_j = \big(\sum_{i=1}^{l} x_{i,j}\big)^{\frac{1}{p}}$.

$$\Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} x_{i,j}\big)^{\frac{1}{p}}}{m}\Big)^{p} = \Big(\frac{\sum_{j=1}^{m} y_j}{m}\Big)^{p} = \frac{1}{m^p}\Big(\sum_{j=1}^{m} y_j\Big)^{p}.$$

Since there are m' matched keywords, there are only m' non-zero y_j. For simplicity, we assume $y_j > 0$, $1 \le j \le m'$. So, $\sum_{j=1}^{m} y_j = \sum_{j=1}^{m'} y_j$.

Then, we apply Jensen's inequality. We have $\frac{\sum_{j=1}^{m'} y_j^p}{m'} \ge \Big(\frac{\sum_{j=1}^{m'} y_j}{m'}\Big)^{p} = \frac{1}{m'^p}\big(\sum_{j=1}^{m'} y_j\big)^{p}$. So, we have $m'^{p-1}\sum_{j=1}^{m'} y_j^p \ge \big(\sum_{j=1}^{m'} y_j\big)^{p}$.

$\Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} x_{i,j}\big)^{\frac{1}{p}}}{m}\Big)^{p} = \Big(\frac{\sum_{j=1}^{m} y_j}{m}\Big)^{p} \le \frac{m'^{p-1}}{m^p}\sum_{j=1}^{m'} y_j^p = \frac{m'^{p-1}}{m^p}\Big(\sum_{j=1}^{m}\sum_{i=1}^{l} x_{i,j}\Big) = \frac{m'^{p-1}}{m^p}\sum_{i=1}^{l} LS_i$.

$\frac{m'^{p-1}}{m^p}\sum_{i=1}^{l} LS_i$ *is a monotonic upper bound of our global score.*

Since m' may be different for different JTTs, we define the following uniform upper bound.

DEFINITION 13. *The monotonic upper bound of the global score is defined as:*

$$upper_{content}(T,Q) = \frac{m_{CN}^{p-1}}{m^p}\sum_{i=1}^{l} LS_i$$

where m_{CN} is the upper bound of m' of all JTTs corresponding to a given CN.

We can compute LS_i for each non-free tuple separately and then obtain a monotonic upper bound $\frac{m_{CN}^{p-1}}{m^p}\sum_{i=1}^{l} LS_i$ for the *global score.*

6. EXECUTION ENGINE ISSUES

The *structure quality* and the *content quality* together determine the effectiveness of keyword search which is certainly the most important factor. The efficiency problem is also very important to user experience. Generating all possible JTTs will induce prohibitively long query time for large databases. So, we need to consider efficiency issues of the execution engine. The general existing execution flowchart is shown in Figure 7.

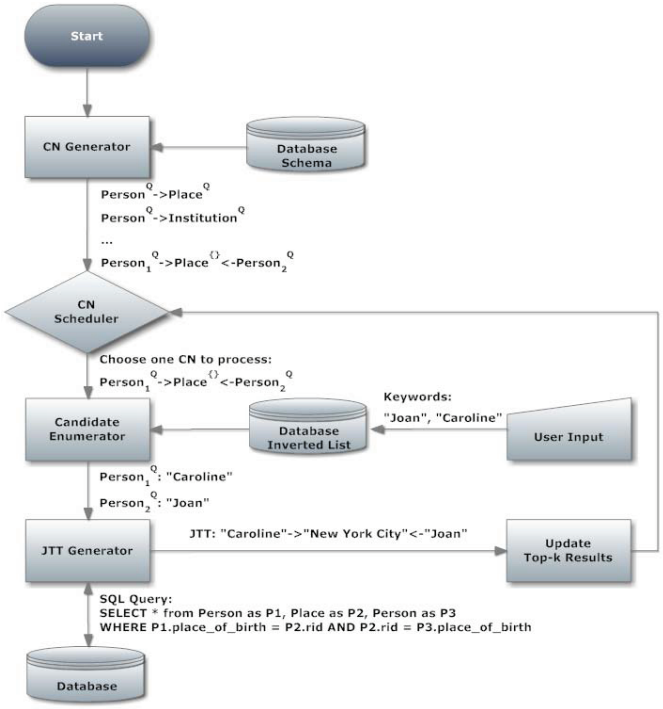


Figure 7: Execution Flowchart

First, the CN generator generates all possible CNs. Assume that we can estimate an upper bound for the highest *result quality* score of all JTTs corresponding to a CN. Each time, the CN scheduler chooses the CN with the highest upper bound. So, we have an opportunity to find a JTT with the highest *result quality* score, corresponding to the chosen CN. Next, for the chosen CN, we enumerate possible JTTs using efficient top-k algorithms (candidate enumerator). For each set of candidates (non-free tuple sets), we then generate all corresponding JTTs by joining candidates together with free tuples (JTT generator). For each generated JTT, we update top-k results. The execution engine stops when the highest CN upper bound is not higher than the current top-k result scores.

In this section, we mainly discuss the following components: the CN generator, the candidate enumerator, and the JTT generator. The CN generator was originally pro-

posed in [6, 5]. However, we observe that we may generate different but essentially isomorphic CNs based on original criterion. So, we need to improve the CN generator to rule out unnecessary isomorphic CNs. For the candidate enumerator, we modify the Block Pipeline Algorithm [10] to incorporate our *structural quality* metric and the *content quality* metric. In addition, our modified version also improves the space efficiency. We also develop a buffer system for the JTT generator to further explore the opportunity to avoid redundancy of querying DBMS.

6.1 The CN Generation

The function of the CN generator is to generate all possible CNs. Completeness and non-redundancy are two main issues to be considered. Existing CN generation algorithm guarantees completeness. However, redundancy induced by isomorphic CNs cannot be avoided. We briefly introduce our generation algorithm here.

Several properties of a valid CN has been proposed in [6, 5]:

1. The number of non-free tuple sets in a CN does not exceed the number of query keywords m. Otherwise, we can always find a sub-tree of CN which contains all keywords.

2. No leaf tuple sets of CN are free. Otherwise, we can always find a sub-tree of CN which contains all keywords.

3. CN does not contain a construct of the form $R \leftarrow S \rightarrow R$. Otherwise, all resulting JTTs would contain the same tuple more than once.

First, we assume there are n non-free tuple sets $R_1^Q, \ldots, R_n^Q$ and m free tuple sets $R_{n+1}^{\{\}}, \ldots, R_{n+m}^{\{\}}$, and a pre-defined maximum CN size. Initially, we have n valid size-1 CNs $R_1^Q, \ldots, R_n^Q$ and we keep them in an expansion list. For every iteration, we pick one CN from the expansion list. We check its validity and try to expand its size by 1 if the current CN size is smaller than the pre-defined maximum CN size. We insert all expanded CNs into the list. Note that we expand the root node at most once. The procedure terminates when the expansion list is empty.

There are several types of isomorphic situations to be considered since the same tuple set can appear multiple times in a CN. Consider the following two CNs: (1) $R_1^Q \rightarrow R_2^Q$ and (2) $R_2^Q \leftarrow R_1^Q$. These two CNs are identical. The difference is that we pick different tuple sets as the root. To avoid this kind of redundancy, we add the first restriction: if we pick tuple set R_i^Q as the root node, tuple sets $R_j^Q, 1 \leq j < i$ are not allowed to be picked as leaf nodes.

Consider the following two CNs: (1) $R_2^Q \leftarrow R_1^Q \rightarrow R_3^Q$ (R_1^Q is the root node) and (2) $R_3^Q \leftarrow R_1^Q \rightarrow R_2^Q$ (R_1^Q is the root node). These two CNs are identical. The only difference is that we either pick tuple set R_2 first or we pick tuple set R_3 first. To avoid this kind of redundancy, we add the second restriction: If we have already expanded a tuple set R_i from a node node and we try to expand it again, tuple sets $R_j, 1 \leq j < i$ are not allowed to be picked as a new node.

Consider the following two CNs: (1) $R_2^Q \leftarrow R_4^Q \leftarrow R_1^Q \rightarrow R_4^Q \rightarrow R_3^Q$ (R_1^Q is the root node) and (2) $R_3^Q \leftarrow R_4^Q \leftarrow R_1^Q \rightarrow R_4^Q \rightarrow R_2^Q$ (R_1^Q is the root node). These two CNs are identical. This kind of situation cannot be eliminated by previous two restrictions. To avoid this kind of redundancy, we need to do a post-check. Specifically, we use the smallest representation of a CN tree to eliminate redundancy. One possible solution is as follow:

1. For a leaf node, e.g. R_2^Q, its smallest representation is a string "(R_2)".

2. If a subtree root R_k^Q has l children and its children's smallest representations are $rep_1, rep_2, \ldots, rep_l$. We sort these children's smallest representations in lexicographic order, $str_1, str_2, \ldots, str_l$. Then, the smallest representation of the subtree is "$(R_k$" $+ str_1 + \ldots +$ $str_l +$ "$)$".

3. The smallest representation of a CN can be computed recursively in a bottom-up fashion.

Consider the situation mentioned above, the smallest representation of CN: $R_2^Q \leftarrow R_4^Q \leftarrow R_1^Q \rightarrow R_4^Q \rightarrow R_3^Q$ is "$(R_1(R_4(R_2))(R_4(R_3)))$" and the smallest representation of CN: $R_3^Q \leftarrow R_4^Q \leftarrow R_1^Q \rightarrow R_4^Q \rightarrow R_2^Q$ is also "$(R_1(R_4(R_2))(R_4(R_3)))$". Based on the smallest representation, we know these two CNs are essentially identical.

With the three restrictions above, we can eliminate redundancy of CN generation.

6.2 The Candidate Enumerator

For a chosen CN, the candidate enumerator choose a candidate (a candidate is a set of non-free tuples) corresponding to the given CN. Since the *result quality*, $score(T, Q) = f_{structural}(T) \times f_{content}(T, Q)$, where $f_{structural}(T)$ is determined by the chosen CN and $f_{content}$ is determined by non-free tuples (candidate), $score(T, Q)$ is fully determined by the chosen candidate. Therefore, candidate enumerator chooses the candidate with the highest possible $score(T, Q)$, which is also the score upper bound of the chosen CN.

To improve efficiency, top-k algorithms (for example, the Skyline Sweeping Algorithm and the Block Pipeline Algorithm) can be used in this component. Most top-k algorithms work based on the following assumption: the *result quality* score is a monotonic function of numeric local scores of individual tuples. Note that definitions of local scores are different in different existing approaches.

Suppose there are n non-free tuple sets $R_1^Q, \ldots, R_n^Q$ in a CN. All non-free tuple sets are sorted by their local scores in descending order, resulting n sorted lists, $list_1, \ldots, list_n$. Our enumeration space is $SPACE_{CN} = list_1 \times \ldots \times list_n$. Then, a candidate is an element in $SPACE_{CN}$. Suppose candidate $C_i = (t_{i_1}, \ldots, t_{i_n}) \in SPACE_{CN}$. We use $res(C_i)$ to denote the set of all corresponding results. Since the *result quality* score is determined by the chosen candidate, the *result quality* scores of all results in $res(C_i)$ are the same.

Suppose $C_i = (t_{i_1}, \ldots, t_{i_n}), C_j = (t_{j_1}, \ldots, t_{j_n})$. We define a partial order on $SPACE_{CN}$: $C_i \leq C_j$ if $i_1 \leq j_1, \ldots, i_n \leq j_n$. Suppose the *result quality* score is monotonic. If $C_i \leq C_j$, the *result quality* scores of all results in $res(C_i)$ is not lower than the *result quality* scores of all results in $res(C_j)$. So, we should choose candidates according to their partial order. Therefore, skyline algorithms are useful in this case.

However, the *result quality* score is not monotonic due to *content quality* is not monotonic. Fortunately, we have a monotonic upper bound for the *global score* of *content quality* (see Section 5.2). If $C_i \leq C_j$, we can guarantee that $upper_{content}(a, Q) \geq upper_{content}(b, Q)$, $score_{structural}(a) = score_{structural}(b)$ for any $a \in ans(C_i), b \in ans(C_j)$. With the monotonic upper bound of *result quality*, $upper(T, Q) = upper_{content}(T, Q) \times score_{structural}(T)$, skyline algorithms are still useful.

Then, we will briefly introduce our revised version of the Block Pipeline Algorithm, which is originally proposed in [10].

BP Block Pipeline Algorithm
input: all CNs and query; **return**: top-k answers

```
Queue ← ∅
for all CNs do
    b ← the first block of CN
    b.status ← USCORE
    Queue.push(b, calc_u score(b))
end for
while topk[k].score < Queue.head().getscore() do
    head ← Queue.pop_max()
    if head.status = USCORE then
        head.status ← BSCORE
        Queue.push(head, calc_b score(head))
        for all unvisited adjacent blocks b' do
            b'.status ← USCORE
            Queue.push(b', calc_u score(b'))
        end for
    else if head.status = BSCORE then
        res ← execute(b)          ▷ enumerate/check candidate
        for all result T ∈ res do
            T.status ← SCORE
            Queue.push(T, calc_s core(T))
        end for
    else
        Insert head into topk
    end if
end while
return topk
```

The intuition of Skyline Algorithms is that if there are two candidates T_x and T_y from the same CN and $upper(T_x, Q) \geq upper(T_y, Q)$, T_y should not be enumerated unless T_x has been enumerated. We define a dominant relationship among candidates. Denote $T_x.d_i$ as the order of candidate T_x on the non-free tuple set R_i^Q. If $T_x.d_i \leq T_y.d_i$ for each non-free tuple set R_i^Q, then $upper(T_y, Q) \leq upper(T_x, Q)$. After

we enumerate a candidate T_x, we push all other candidates directly dominated by T_x into a priority queue by the descending order of their upper bound scores. The algorithm stops when the real score of the current top-kth result is not less than the upper bound score of the head element of the priority queue; the latter is exactly the upper bound score of the entire unvisited enumeration space.

In our revised version of the Block Pipeline Algorithm, we adopt the signature definition of a tuple t from the Block Pipeline Algorithm ([10]). Denote the signature of a tuple t as $sig(t) = (tf_{w_1}(t), \ldots, tf_{w_m}(t))$, which is an ordered sequence of term frequencies for all the query keywords $w_1, \ldots, w_m$.

By replacing the document length term with the minimal document length, we define $tfidf_B(t, w_i) \geq tfidf(t, w_i)$ and $score_B(T, Q) \geq score(T, Q)$ as follows.

$$tfidf_B(t, w_i) = \begin{cases} 0 & tf_{w_i}(t) = 0 \\ \frac{1+\log(1+\log(tf_{w_i}(t)))}{(1-s)+s\frac{mindl}{avdl}} \cdot \log(idf_{w_i}) & tf_{w_i}(t) > 0 \end{cases}$$

$$score_B(T, Q) = C(CN)\Big(\frac{\sum_{j=1}^{m}\big(\sum_{i=1}^{l} PG(t_i) tfidf_B(t_i, w_j)\big)^{\frac{1}{p}}}{m}\Big)^p$$

Then, we define the signature of a candidate as a matrix $sig(T) = (sig(t_1), sig(t_2), , sig(t_k))$ which is different than the original Block Pipeline Algorithm. For a given CN, the partitioning of its non-free tuple sets naturally incurs a partitioning of all candidates. Each partition of candidates is called a block. All candidates within the same block have the same signature and the same non-monotonic upper bound $score_B(T, Q)$.

Then, we define a new monotonic upper bound for the *final score* $upper_B(T, Q) = C(CN)\frac{m_{CN}^{p-1}}{m^p} LS_B^i \geq upper(T, Q)$ where $LS_B^i = \sum_{j=1}^{m} PG(t_i) \cdot score_B(t_i, w_j) \geq S_i$. If two candidates have the same signature, they have the same upper bound $upper_B(T, Q)$. All candidates within the same block have the same monotonic upper bound $upper_B(T, Q)$ and non-monotonic upper bound $score_B(T, Q)$. So, we use $upper_B(T, Q)$ as $uscore(b, Q)$ of a block and we use $score_B(T, Q)$ as $bscore(b, Q)$ of a block in our revised version of the Block Pipeline Algorithm.

Note that, in the worst case, the total number of blocks can be very large. Suppose that each non-free tuple set can be divided into x strata and the number of keywords is m. The total number of blocks of a CN can be as large as x^m. Then, we have a problem: the maximum size of the priority queue can be as large as $O(|CN| \cdot x^m)$. We may not have enough space to store the entire priority queue in the main memory.

Note that there are three kinds of elements in the priority queue: $USCORE$, $BSCORE$, $SCORE$ (the real *result quality* socre). For all candidates, $USCORE \geq BSCORE \geq SCORE$. Among them, all $USCORE$ elements are necessary for correctness because we have to store the entire skyline of the chosen CN to guarantee that all unvisited candidates are upper bounded. We only need to store k $SCORE$ elements because we only need top-k results. It is unnecessary to store a $SCORE$ element if there exist k better results. The number of $BSCORE$ elements can be flexible. The intention of the Block Pipeline Algorithm is to enumerate (check) candidates in a lazy fashion. If we enumerate (check) a candidate early, it will not affect correctness.

Note that the length of an arbitrary skyline is at most x^{m-1}. So, the maximum number of $USCORE$ elements at any time is $|CN| \cdot n^{m-1}$ which is much smaller than the total number of blocks. To guarantee correctness, we must keep all $USCORE$ elements, k SCORE elements, and at least 1 $BSCORE$ element. However, to improve efficiency, we should keep as many $BSCORE$ elements as possible.

We break the priority queue into three priority queues and put different size restrictions on them. We may now check the head element of the $BSCORE$ priority queue to decrease the queue size when the queue is full. Although we may check more $BSCORE$ elements, we can produce $SCORE$ elements faster and, hence, generate a higher bound for top-k results. With a higher bound, we do not need to keep any $BSCORE$ less than the current bound. Hence, the space problem can be controlled.

If the maximum number of $USCORE$ elements is still too large, we have the following compression method to solve the space problem. We can combine a group of adjacent strata into one super-stratum. For a given CN, the partitioning of super-strata also incurs a partitioning of all candidates. Each partition of candidates is called a super-block. We assign the highest $upper_B(T, Q)$ in a super-block as $uscore(b, Q)$ of the super-block and we assign the highest

$score_B(T, Q)$ in a super-block as $bscore(b, Q)$ of the super-block. Therefore, the space problem is solved.

Note that the efficiency of the algorithm largely depends on the quality of $uscore(b, Q)$. If the bound of $uscore(b, Q)$ is too loose, the algorithm will enumerate all blocks and calculate its $bscore(b, Q)$. Therefore, the choice of scoring function also affects runtime efficiency.

6.3 The JTT generator

The purpose of the JTT generator is to generate all possible JTTs according to a specific candidate ($execute(b)$ in the pseudocode). Existing implicit data graph approaches will simply generate a single complex SQL query and throw the question to the underlying DBMS.

It is very inefficient to issue a single SQL query for each candidate because executing a SQL query and fetching its results require inter-process communications, and suffer from DBMS internal overheads (for example, parsing and planning). Since join selectivity may be very low and, hence, most of such probing queries will return an empty set, a large number of probing queries will be sent to the underlying DBMS to get top-k results. We can improve efficiency by using range parametric queries. The Global Pipeline Algorithm ([5]), the Skyline Sweeping Algorithm ([10]), and the Block Pipeline Algorithm ([10]) all benefit from grouping candidates into ranges and issue a single parametric query to the database. However, improvements are possible.

Most candidates of the same CN partially overlap each other. Consider the following example, two keywords *'New York'* and *'Harvard'*. Given a CN: $Place^Q \leftarrow Person^{\{\}} \rightarrow Institution^Q$, we may have two candidates: (1) 'Southampton, New York' ($rid = p3$) and 'Harvard University' ($rid = e2$); (2) 'New York City, New York' ($rid = p4$) and 'Harvard University' ($rid = e2$). For these two candidates, existing approaches will issue the following two SQL queries:

1. ```
   SELECT * FROM Place, Person, Institution
   WHERE Place.rid = p3 AND Institution.rid = e2
   AND Person.place_of_birth = Place.rid
   AND Person.Alma_mater = Institution.rid
   ```

2. ```
   SELECT * FROM Place, Person, Institution
   WHERE Place.rid = p4 AND Institution.rid = e2
   AND Person.place_of_birth = Place.rid
   AND Person.Alma_mater = Institution.rid
   ```

Since these two SQL queries partially overlap, we can reuse the common intermediate result. We issue a single query: `SELECT * FROM Person WHERE Person.Alma_mater = e2`. Then we need to check each tuple's `place_of_birth` attribute, respectively.

We can improve efficiency by exploring reusability. Existing approaches communicate with DBMS at the JTT level (one SQL query to retrieve a set of JTTs). In contrast, we communicate with DBMS at the tuple level (one SQL query to retrieve either one tuple or a set of tuples). We issue only the following two types of simple SQL queries:

1. ```
   SELECT * FROM REL_A WHERE REL_A.A_id = XXX
   ```

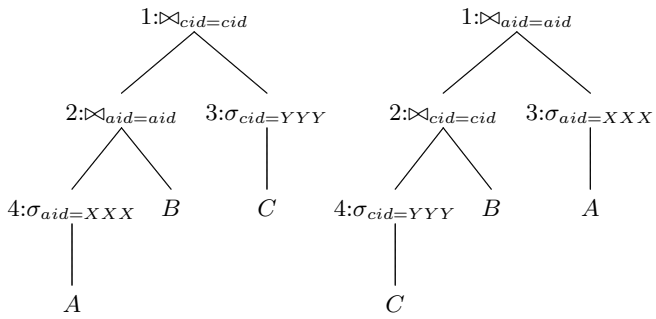
2. ```
   SELECT * FROM REL_A WHERE REL_A.B_id = XXX
   ```

A type 1 SQL query will return exactly one result because `A_id` is `REL_A`'s primary key. Type 2 SQL query may return multiple results because `B_id` is a foreign key. Since we have built hash indices on all primary keys and foreign keys, these two types of SQL queries can be executed efficiently by the underlying DBMS. We then discuss how to check existence of JTTs efficiently by using these two types of simple SQL queries.

Existing approaches throw the complex SQL queries to the underlying DBMS. The optimizer of the DBMS will choose an execution plan for a given SQL query and the optimizer will choose the best execution plan for the current candidate. However, the best execution plan for the current candidate might not be the best plan for the entire CN if we take reusability into consideration.

Consider the following example. Given a CN: $A^Q \leftarrow B^{\{\}} \rightarrow C^Q$, suppose there are 100000 tuples in B, 10000 tuples in A, 20000 tuples in C, and $A.aid$, $C.cid$ are primary keys. In addition, we have hash indices on $A.aid$, $B.aid$, $B.cid$, and $C.cid$. We assume that each tuple in A is referred by 10 tuples in B ($A.aid = B.aid$) and each tuple in C is referred by 5 tuples in B ($C.cid = B.cid$).

We have two possible execution plans:



Because $A.aid$ and $C.cid$ are primary keys, node 4 and node 3 will retrieve exactly one tuple. The I/O costs of nodes 1, node 3, and node 4 are exactly the same. The only difference is the I/O cost of node 2. It depends on how many tuples will be retrieved on node 2. For a given candidate ($a : XXX$, $c : YYY$), the latter plan is better. However, the latter plan might not be better than the former one for the entire CN.

We assume that the size of non-free tuple set A^Q is 10 and the size of non-free tuple set C^Q is 1000. If we can re-use retrieved tuples, the total I/O cost for the entire CN will be determined by the total number of tuples in B will be retrieved. If we use the former plan, $10 \times 10 = 100$ tuples in B will be retrieved in the worst case. If we use the latter plan, $1000 \times 5 = 5000$ tuples in B will be retrieved in the worst case. The former plan is much better than the latter one. To test $10 \times 1000 = 10000$ candidates, we only need to retrieve $10 \times 10 = 100$ tuples in B by issuing 10 simple SQL queries (`SELECT * FROM B WHERE B.aid = XXX`). To choose such an appropriate plan for generating JTTs according to a given CN, we need a special optimizer beyond the DBMS. One possible solution of the optimizer is discussed in the appendix.

6.4 The Buffer System

The efficient JTT generation algorithm works based on an assumption that we do not need to retrieve the same tuple repeatedly. The support of a buffer system is necessary. In addition, our buffer system gives us new opportunities to improve efficiency. First, two different CNs may overlap (for example, the left part of the following two CNs are the same: $Person^Q \xleftarrow{father-child} Person^{\{\}} \xrightarrow{child-mother} Person^Q$ and $Person^Q \xleftarrow{father-child} Person^{\{\}} \xrightarrow{spouse-spouse} Person^Q$). Second, two queries may partially overlap (for example, two keywords query *'Caroline'* and *'John'* and two-keywords query *'Caroline'* and *'Joseph'*).

If the buffer pool is large enough, we should not issue the same SQL query twice. We should check whether a SQL query has been issued in an efficient way. Therefore, we organize our buffer pool as a large hash table. Since dynamic memory allocation or re-allocation is time consuming, entries of the hash table should have a fixed size.

Note that type 1 SQL queries (primary key) will return exactly one tuple. However, type 2 SQL queries (foreign key) will return a tuple set which may contain multiple tuples. Since the size of entries is fixed, we cannot store the entire result of a type 2 SQL query in a single entry. So, a type 2 SQL query may occupy multiple entries. Text attributes are not necessary for JTT generation algorithms. We only need to keep primary key and foreign keys of a tuple. Hence, a tuple can be stored as an array of keys. We can store a type 1 SQL query in a single entry. Next, we discuss how to store a type 2 SQL query.

A naive method is that we store a type 2 SQL in consecutive entries. First, we find the head entry and store the size of result (the number of occupied entries) with the head entry. We store the rest of the result in the following entries. However, this organization is not good enough for the following two reasons:

1. A consecutive allocation may induce many more collisions. We may have the following situation: The total free space of the buffer pool is large enough. However, we cannot find a large enough consecutive free space.

2. A tuple may be stored multiple times in the buffer pool. The maximum number of duplicates is the number of foreign keys.

These two problems harm the space efficiency of the buffer system. We propose a new design for the buffer system: a hash table with cursors. The intuition is that we only keep a unique copy for each tuple in the buffer pool. The position of the entry for a tuple can be uniquely identified by its primary key. For example, a tuple $T_A \in A$ can be uniquely identified by $(A, T_A.A\_id)$.

Given the result of a type 2 SQL query, the positions of entries for the tuples are already determined by their primary

keys. We know where to store these tuples. In addition, we do not require consecutive data space. However, there is a new problem for retrieving the type 2 SQL query result because these tuples are scattered in the buffer pool.

Originally, in each entry, we have an array of keys (primary key and foreign keys). We add an array of "head" cursors. Consider the following type 2 SQL query: `SELECT * FROM A WHERE A.B_id = X`. Let T_B denote tuple (B, X). So, $T_B.B\_id = X$. Initially, we set $T_B.HEAD\_A = NULL$. If the result set is not empty and T_A $(A, T_A.A\_id)$ is the first tuple in the result set, we set $T_B.HEAD\_A = T_A$. If the result set is empty, we set set $T_B.HEAD\_A = T_B$. Therefore, we determine whether we have the result for the given type 2 SQL query by the "head" cursor $T_B.HEAD\_A$. If the cursor points to $NULL$, we do not have the result in our buffer pool. If the cursor points to T_B itself, we know the result set is empty. Otherwise, the cursor points to the first tuple in the result set. Note that a tuple may have multiple "head" cursors. The number of "head" cursors of a tuple is equal to the number of relations which are referring to the relation of the tuple.

Then, we add another array of "next" cursors to retrieve the rest of the tuples in the result set. Consider the same example: `SELECT * FROM A WHERE A.B_id = X`. Now, we can find the first tuple in the result T_{A_1} $(A, T_{A_1}.A\_id)$ by checking the head cursor $T_B.HEAD\_A$. Suppose the second tuple is T_{A_2}. We set $T_{A_1}.NEXT\_B = T_{A_2}$. Similarly, we set $T_{A_i}.Next\_B = T_{A_{i+1}}$ if the next tuple is $T_{A_{i+1}}$. We set $T_{A_j}.NEXT\_B = NULL$ if T_{A_j} is the last tuple. In this way, all tuples in the result set are linked together (like a linked list). We can retrieve them one by one through the "next" cursors. Also note that a tuple may have multiple "next" cursors. The number of "next" cursors of a tuple is equal to the number of foreign keys of the tuple.

Therefore, based on the original array of keys (primary key and foreign keys), we just need to add at most D cursors (also keys), where D is the maximum degree of SG.

Note that the buffer pool might not be large enough to store all retrieved tuples. We still need to choose a replacement policy. The Least Recently Used replacement policy is perfect for our purpose. Therefore, we keep a dynamic partial data graph in the buffer pool. The dynamic partial data graph is managed by our buffer system. We try to keep the most recent "hot" tuples. Hence, these "hot" tuples can be retrieved very fast (no communications with DBMS, no extra I/O cost). In the long run, our system can be as efficient as those explicit data graph approaches. In addition, our system also works on very large databases as those implicit data graph approaches.

7. EXPERIMENT RESULT

We use the DBLP dataset to evaluate effectiveness and efficiency of our approach and existing approaches. DBLP dataset is a XML file. It is not a typical relational database. So, we adopt the database schema used in [10] and convert the DBLP XML database into a relational database. The relation schemas and statistics of the dataset can be found in Table 3.

We manually pick 9 sets of queries, which cover different selectivity of keywords (some keyword only matches about 10 tuples and some keyword matches more than 5000 tuples), different size of answers, different relationships (CN), and different top-k settings.

We use PostgreSQL v8.3 with default configuration as the underlying DBMS. We implement the Sparse Algorithm (SP), the Global Pipeline Algorithm (GP), the Block Pipeline Algorithm (BP), and our approach - the Universal Algorithm (UA). We have improved the GP and the BP algorithms by using range parametric query optimization. All algorithms are implemented using GCC 4.3. All experiments were run on a PC with an Intel Core2 Duo 1.83GHz CPU and 3.00G memory running Ubuntu 8.04. The database server and the client are on the same PC.

7.1 Effectiveness

We compare the following four approaches: the Sparse Algorithm (Sparse), the Global Pipeline Algorithm (GP), the Block Pipeline Algorithm (BP), and the Universal Algorithm (UA).

Our experiment shows that the Sparse with OR-semantics, the GP with OR-semantics can hardly produce any expected results. In most cases, none of the top-k answers contains all keywords. So, we force AND-semantics for these two algorithms. For the BP algorithm, the default tuning parameter

Relation Schema	# of tuples
Author(AuthorID, *Name*)	665827
InProceeding(InProceedingID, *Title*, ProceedingID)	675638
Proceeding(ProceedingID, *Title*, PublisherID, SeriesID)	11216
Series(SeriesID, *Title*)	86
Publisher(PublisherID, *Name*)	385
Writes(WID, AuthorID, InProceedingID)	1799039

Table 3: DBLP dataset

2.0 (used in [10]) is not large enough to produce good results. Hence, we choose the tuning parameter $p = 64.0$ for the BP algorithm to improve effectiveness. For the UA, we choose the default tuning parameter $p = 3.0$. Note that we terminate an algorithm if it cannot produce answers in 2000 seconds.

7.1.1 Query 1: 'tamer' and 'databases', top-10 answers

Results are shown in Table 4, Table 5, and Table 6. GP exceeds the time limit (2000 seconds). The output of other three approaches are acceptable and both keywords are matched. Note that all answers correspond to the same CN: $Author^{Q} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{Q}$ (an author 'tamer' writes some paper about 'databases'). All answers are acceptable.

7.1.2 Query 2: 'nikos' and 'clique', top-1 answer

Results are shown in Table 7. This test query was used in [10]. The output of all four approaches are identical. The only answer corresponds to the same CN: $Author^{Q} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{Q}$ (an author 'nikos' writes some paper about 'clique').

7.1.3 Query3: 'tamer' and 'ihab', top-5 answers

Results are shown in Table 8, Table 9, and Table 10. The output of all four approaches are acceptable and both keywords are matched. Although Rank-1 answers are not the same, all Rank-1 answers correspond to the same CN: $Author^{Q} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{\{\}} \leftarrow Writes^{\{\}} \rightarrow Author^{Q}$ ('tamer' and 'ihab' are co-authors). Therefore, all answers are acceptable.

7.1.4 Query 4: 'qagen' and 'in-network', top-2 answers

Results are shown in Table 11 and Table 12. BP and UA output the same top-2 answers. The Rank-1 answer corresponds to a CN: $InProceeding^{Q} \rightarrow Proceeding \leftarrow InProceeding^{Q}$. The Rank-2 answer corresponds to a CN: $InProceeding^{Q} \leftarrow Writes^{\{\}} \rightarrow Author^{\{\}} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{Q}$ (two papers are written by the same author). Sparse and GP output a different Rank-2 answer corresponding to a CN: $InProceeding^{Q} \rightarrow Proceeding^{\{\}} \rightarrow Publisher^{\{\}} \leftarrow Proceeding^{\{\}} \leftarrow InProceeding^{Q}$ (two papers are accepted by two different proceedings respectively and these two proceedings are published by the same publisher). Obviously, BP and UA output a better Rank-2 answer. So, on this test query, BP and UA are winners.

7.1.5 Query 5: 'ihab' and 'sigmod', top-10 answers

Results are shown in Table 13 and Table 14. Sparse exceeds the time limit (2000 seconds). The output of GP/BP matches both keywords. However, there is no acceptable answers. These answers all correspond to that the same CN: $Author^{Q} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{\{\}} \rightarrow Proceeding^{\{\}} \rightarrow Publisher^{\{\}} \leftarrow Proceeding^{\{\}} \leftarrow InProceeding^{Q}$. There is no meaningful relationship corresponding to this CN. All these answers share a common postfix: $\rightarrow$ Publisher:ACM $\leftarrow$ Proceedings of the ACM SIGMOD,SIGMOD 2008 $\leftarrow$ Corrigendum to ... (proc.SIGMOD 03). Note that this common postfix contributes three matches of keyword 'sigmod'. This postfix seriously influenced these three algorithms.

In the DBLP database, there are six acceptable answers ('Ihab F. Ilyas' has six papers accepted by 'sigmod'). These acceptable answers all correspond to the same CN: $Author^{Q} \leftarrow$

Rank	Top-10 Answers (Sparse)
1	Tamer Kahveci ← Writes → MAP: Searching Large Genome Databases
2	Tamer Kahveci ← Writes → Fast alignment of large genome databases
...	...
9	M. Tamer Özsu ← Writes → Experimenting with Temporal Relational Databases
10	M. Tamer Özsu ← Writes → DBFarm: A Scalable Cluster for Multiple Databases

Table 4: 'tamer' and 'databases': Sparse

Rank	Top-10 Answers (BP)
1	Rank-1 of Sparse/GP
2	Rank-2 of Sparse/GP
...	...
9	M. Tamer Özsu ← Writes → Database Support for Document and Multimedia Databases
10	M. Tamer Özsu ← Writes → VisualMOQL: A Visual Query Language for Image Databases

Table 5: 'tamer' and 'databases': BP

Rank	Top-10 Answers (UA)
1	Rank-1 of Sparse/GP
2	Rank-2 of Sparse/GP
...	...
9	M. Rank-10 of Sparse/GP
10	M. Tamer Özsu ← Writes → A Multi-Level Index Structure for Video Databases

Table 6: 'tamer' and 'databases': UA

Rank	Top-1 Answer (Sparse / GP / BP / UA)
1	Nikos Mamoulis ← Writes → Constraint-Based Algorithms for Computing Clique Intersection Joins

Table 7: 'nikos' and 'clique': Sparse/GP/BP/UA

Rank	Top-5 Answers (Sparse / GP)
1	Ihab Amer ← Writes → A Hardware-Accelerated Framework with IP-Blocks for Application in MPEG-4 ← Writes → Tamer Mohamed
...	...
4	Ihab F.Ilyas ← Writes → FIX: Feature-based Indexing Technique for XML Documents ← Writes → M. Tamer Özsu
...	...

Table 8: 'tamer' and 'ihab': Sparse/GP

Rank	Top-5 Answers (BP)
1	Rank-4 of Sparse/GP
2	Rank-1 of Sparse/GP
...	...

Table 9: 'tamer' and 'ihab': BP

Rank	Top-5 Answers (UA)
1	Rank-1 of Sparse/GP
2	Rank-2 of Sparse/GP
...	...

Table 10: 'tamer' and 'ihab': UA

Rank	Top-2 Answers (Sparse / GP)
1	In-network execution of monitoring queries in sensor networks → SIGMOD 2007 ← QAGen: generating query-aware test databases
2	Optimization of in-network data reduction → Proceedings of the 1st Workshop on DMSN 2004 → Publisher:ACM ← SIGMOD 2007 ← QAGen: generating query-aware test databases

Table 11: 'qagen' and 'in-network': Sparse/GP

Rank	Top-2 Answers (BP / UA)
1	Rank-1 of Sparse/GP
2	In-network execution of monitoring queries in sensor networks ← Writes → M. Tamer Özsu ← Writes → QAGen: generating query-aware test databases

Table 12: 'qagen' and 'in-network': BP/UA

Rank	Top-10 Answers (GP / BP)
1	Ihab F.Ilyas ← Writes → CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies → Proceedings of the ACM SIGMOD,2004 → Publisher:ACM ← Proceedings of the ACM SIGMOD,SIGMOD 2008 ← Corrigendum to "efficient similarity search and classification via rank aggregation" (proc.SIGMOD 03)
2	Ihab F.Ilyas ← Writes → Rank-aware Query Optimization → Proceedings of the ACM SIGMOD,2004 → Publisher:ACM ← Proceedings of the ACM SIGMOD,SIGMOD 2008 ← Corrigendum to "efficient similarity search and classification via rank aggregation" (proc.SIGMOD 03)
...	...

Table 13: 'ihab' and 'sigmod': GP/BP

Rank	Top-10 Answers (UA)
1	Ihab F.Ilyas ← Writes → CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies → Proceedings of the ACM SIGMOD,2004
2	Ihab F.Ilyas ← Writes → Rank-aware Query Optimization → Proceedings of the ACM SIGMOD,2004
...	...

Table 14: 'ihab' and 'sigmod': UA

$Writes^{\{\}} \rightarrow InProceeding^{\{\}} \rightarrow Proceeding^{Q}$. The Top-6 answers in the output of UA are exactly these acceptable answers. Since our approach uses a more reasonable *coherence score* for the *structural quality*, those meaningless CNs will be assigned a relatively low *coherence score* and, hence, our approach is less influenced by the long postfix. Therefore, on this test query, UA is the only winner.

7.1.6 Query 6: 'ihab' and 'query', top-5 answers

Results are shown in Table 15 and Table 16. Spase and GP both exceed the time limit (2000 seconds). None of the top-5 answers in the output of BP are acceptable. There are two kinds of CNs: $Author^{Q}$ and $Author^{Q} \leftarrow Writes^{\{\}} \rightarrow InProceeding^{\{\}} \rightarrow Proceeding^{\{\}} \rightarrow Publisher^{\{\}} \leftarrow Proceeding^{Q}$. The former one cannot match both keywords and the latter one is meaningless. The reason why BP chooses these two kinds of answers is as follows. Since we have 13 matched authors, 4909 matched papers, and 5 matched proceedings, the inverse document frequency of $Author^{Q}$ and $Proceeding^{Q}$ is much higher than $InProceeding^{Q}$. Hence, BP tends to match keywords $Author^{Q}$ and $Proceeding^{Q}$. This problem influences the ranking of BP.

All top-5 answers in the output of UA are acceptable answers. Our approach can produce acceptable answers for the following two reasons: (1) our *global score* for the *content quality* penalizes answers which match only one keyword properly; (2) our *coherence score* for the *structural quality* penalizes meaningless CN properly. Therefore, on this test query, UA is the only winner.

7.1.7 Query 7: 'tamer' and 'ihab' and 'ning', top-3 answers

Results are shown in Table 17 and Table 18. Sparse, GP, and UA all output acceptable answers while BP output no acceptable answers. All acceptable answers correspond to the same CN (three co-authors). The size of this CN is seven. Sparse and GP can produce acceptable answers due to forcing AND-semantics (the cost is poor efficiency). Our approach can produce acceptable answers because we identify the size-7 co-author relationship as a close relationship based on its *coherence score*. So, on this test query, Sparse, GP and UA are winners.

7.1.8 Query 8: 'tamer' and 'indexes' and 'ihab', top-15 answers

Results are shown in Table 19 and Table 20. The Rank-1 answer in the output of UA is the only acceptable answer. However, we are searching for the top-15 answers. Inevitably, there are a lot of unexpected answers in the output. If we force AND-semantics, the system would exhaustively search the whole database. Hence, Sparse and GP exceed the time limit (2000 seconds). Observe that the Rank-1 answer in the output of BP is the left half of the acceptable answer and the Rank-2 answer in the output of BP is the right half of the acceptable answer. However, no complete acceptable answer in the output of BP. So, on this test query, UA is the only winner.

7.1.9 Query 9: 'succinct' and 'xbench' and 'multi-scale', top-2 answers

Results are shown in Table 21, Table 22, and Table 23. UA output two acceptable answers. The Rank-1 answer corresponds to the relationship: three papers are accepted by the same proceeding. The Rank-2 answer corresponds to the relationship: three papers are written by the same author. Sparse, GP and BP only find one acceptable answer (Their Rank-2 answers are meaningless). So, on this test query, UA is the only winner.

7.2 Efficiency

We compare the following four approaches: Sparse (AND-semantics), GP (AND-semantics), BP ($p = 64.0$), and UA ($p = 3.0$). We use underlined numbers to denote the fastest approach for each test query. Note that UA outperforms other three approaches, especially on hard queries (Query 6, Query 7. Query 8).

7.3 Summary

Nine test queries can be divided into three levels: (1) Query 1, Query 2, and Query 3 are two-keywords simple queries; (2) Query 4, Query 5, Query 6 are two-keywords hard queries; (3) Query 7, Query 8, Query 9 are three-keywords queries.

For the first level, all four approaches output acceptable answers. However, for the second level and the third level,

Rank	Top-5 Answers (BP)
1	Ihab Amer
2	Ihab Hamadeh ← Writes → Toward a Framework for Forensic Analysis of Scanning Worms → ETRICS 2006 → Publisher:Springer ← Flexible Query Answering Systems,7th International Conference,FQAS 2006
3	Ihab Hamadeh ← Writes → Toward a Framework for Forensic Analysis of Scanning Worms → ETRICS 2006 → Publisher:Springer ← Flexible Query Answering Systems,6th International Conference,FQAS 2004
4	Ihab Hamadeh ← Writes → Toward a Framework for Forensic Analysis of Scanning Worms → ETRICS 2006 → Publisher:Springer ← Flexible Query Answering Systems,Third International Conference,FQAS 98
5	Ihab Kazem

Table 15: 'ihab' and 'query': BP

Rank	Top-5 Answers (UA)
1	Ihab F.Ilyas ← Writes → Rank-aware Query Optimization
2	Ihab F.Ilyas ← Writes → Rank-Aware Query Processing and Optimization
3	Ihab F.Ilyas ← Writes → Top-k Query Processing in Uncertain Databases
4	Ihab F.Ilyas ← Writes → Nile: A Query Processing Engine for Data Streams
5	Ihab F.Ilyas ← Writes → Estimating Compilation Time of a Query Optimizer

Table 16: 'ihab' and 'query': UA

Rank	Top-3 Answers (Sparse / GP / UA)
1	M. Tamer Özsu ← Writes → XSEED: Accurate and Fast . . . ← Writes → Ihab F.Ilyas ↑ Writes → Ning Zhang
2	M. Tamer Özsu ← Writes → InterJoin: Exploiting Indexes . . . ← Writes → Ihab F.Ilyas ↑ Writes → Ning Zhang
3	M. Tamer Özsu ← Writes → FIX: Feature-based Indexing . . . ← Writes → Ihab F.Ilyas ↑ Writes → Ning Zhang

Table 17: 'tamer' and 'ihab' and 'ning': Sparse/GP/UA

Rank	Top-3 Answers (BP)
1	Ning Ning
2	SPIN-ning Software Architectures: A Method for Exploring Complex
3	Ihab Amer

Table 18: 'tamer' and 'ihab' and 'ning': BP

Rank	Top-3 Answers (BP)
1	Ihab F.Ilyas ← Writes → InterJoin: Exploiting Indexes and Materialized Views in XPath Evaluation
2	M. Tamer Özsu ← Writes → InterJoin: Exploiting Indexes and Materialized Views in XPath Evaluation
. . .	. . .

Table 19: 'tamer' and 'indexes' and 'ihab': BP

Rank	Top-3 Answers (UA)
1	Ihab F.Ilyas ← Writes → InterJoin: Exploiting Indexes . . . ← Writes → M. Tamer Özsu
. . .	. . .

Table 20: 'tamer' and 'indexes' and 'ihab': UA

Rank	Top-2 Answers (Sparse / GP)
1	Succinct Physical Storage Scheme . . . → ICDE 2004 ← Multi-Scale Histograms for . . .
	↑
	XBench Benchmark and Performance Testing of XML DBMSs
2	Succinct Text Indexes . . . → TAMC 2006 ← Time Series Predictions Using Multi-scale Support Vector Regressions
	↓
	Publisher: Springer ← VLDB 2002 Workshop EEXTT ← XBench - A Family . . .

Table 21: 'succinct' and 'xbench' and 'multi-scale': Sparse/GP

Rank	Top-2 Answers (BP)
1	Rank-1 of Sparse/GP
2	Succinct Physical Storage Scheme . . . → ICDE 2004 ← XBench Benchmark and Performance . . .

Table 22: 'succinct' and 'xbench' and 'multi-scale': BP

Rank	Top-2 Answers (UA)
1	Rank-1 of Sparse/GP
2	Succinct Physical . . . ← Writes → M. Tamer Özsu ← Writes → Multi-Scale Histograms for Answering Queries . . .
	↑
	Writes → XBench Benchmark and Performance Testing of XML DBMSs

Table 23: 'succinct' and 'xbench' and 'multi-scale': UA

	Sparse (sec)	GP (sec)	BP (sec)	UA (sec)
Query 1	13.29	> 2000	3.92	0.58
Query 2	5.72	291.10	2.31	0.48
Query 3	0.53	9.09	4.49	0.53
Query 4	33.81	37.54	2.49	0.52
Query 5	> 2000	1207.87	2.72	0.49
Query 6	> 2000	> 2000	9.62	0.53
Query 7	66.62	510.91	25.55	0.55
Query 8	> 2000	> 2000	28.47	0.55
Query 9	35.98	1026.36	4.17	0.64

Table 24: Execution time: Sparse/GP/BP/UA

the result quality of UA is much better than other approaches. On Query 4, BP and UA are winners. On Query 7, Sparse, GP and UA are winners. On Query 5, Query 6, Query 8, and Query 9, UA is the only winner. So, UA is the best approach in terms of effectiveness. In addition, UA is also the fastest approach among all four approaches.

8. CONCLUSION

We thoroughly study the *structural quality* and the *content quality* of JTT. We analyze the influence of these two different quality to the top-k ranking. By summarizing advantages and disadvantages of existing approaches, we adopt insightful ideas from previous work and we also overcome the drawbacks of existing approaches. Experiment result shows that our scoring function is the best in terms of effectiveness.

To improve the runtime efficiency of existing implicit data graph approaches and the space efficiency of existing explicit data graph approaches, we design a hybrid architecture. We improve runtime efficiency by reducing redundancy among the processing of different candidates. To reuse intermediate results and avoid redundant computation, we develop a buffer system to support our candidate enumeration algorithm. In addition, we also choose an execution plan for the entire CN to minimize the total I/O cost for the CN. Experiments also show that our system is more efficient than existing ones.

Our universal top-k keyword search system works effectively and efficiently under a variety of settings.

9. REFERENCES

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

[2] G. Bhalotia, A. Hulgeri, C. Nakhey, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. ICDE, 2002.

[3] S. Brin, L. Page, R. Motwami, and T. Winograd. The pagerank citation ranking: bringing order to the web. ASIS, 1998.

[4] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. SIGMOD, 2007.

[5] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. VLDB, 2003.

[6] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. VLDB, 2002.

[7] F. K. Hwang, D. S. Richards, and P. Winter. The steiner tree problem. Annals of Discrete Mathematics, 1992.

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

[9] F. Liu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. SIGMOD, 2006.

[10] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k keyword query in relational databases. SIGMOD, 2007.

APPENDIX

A. CONNECTIVITY SCORE COMPUTATION

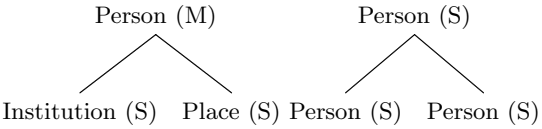
We first define a computation tree corresponding to a given CN. We create a tree node for each tuple set and create tree edges by ignoring all edge directions in SG. Then, we select any one node as the root.

Next, we introduce the definition of types of tree nodes. We recursively define the type of a tree node: single connection node/multiple connection node. There exists, at most, one tuple in a single connection node connecting picked tu-

ples. There may exist more than one tuple in a multiple connection node connecting picked tuples.

DEFINITION 14. *All nodes corresponding to non-free tuple sets are single connection nodes. If the tuple set corresponding to a tree root has an in-edge from the tuple set corresponding to a child node and the child node is a single connection node, then the root is also a single connection node. Otherwise, the root is a multiple connection node.*

For example, there are two CNs: $Institution \leftarrow Person \rightarrow Place$, $Person \xrightarrow{child-father} Person \xleftarrow{father-child} Person$). We can construct two corresponding computation trees as follows (M: multiple, S: Single):



Suppose the tuple set corresponding to the tree root is $R = \{r_1, r_2, \ldots, r_{|R|}\}$. Then, $CS(CN)$ is equal to the probability that r_1 connects all given tuples, or r_2 connects all given tuples, ..., or $r_{|R|}$ connects all given tuples. If R is a single connection node, $CS(CN) = |R| \cdot Pr\{r_i \in R$ connects all given non-free tuples$\}$. Otherwise, $CS(CN) = 1 - (1 - Pr\{r_i \in R$ connects all given non-free tuples$\})^{|R|}$. Let $CS(r_i)$ denote $Pr\{r_i \in R$ connects all given non-free tuples$\}$ and we recursively calculate $CS(r_i)$.

Suppose the tree root has l children. The corresponding tuple sets are $C_1, C_2, \ldots, C_l$. Then, the given CN can be divided into $l+1$ parts, the tuple set corresponding to the root and l subtrees. Therefore, $Pr\{r_i \in R$ connects all given non-free tuples$\} = Pr\{r_i$ connects $c_{1,1}$, $c_{1,1}$ connects all given non-free tuples in the subtree or r_i connects $c_{1,2}$, $c_{1,2}$ connects all given non-free tuples in the subtree or $\ldots\} * \ldots *$
$Pr\{r_i$ connects $c_{l,1}$, $c_{l,1}$ connects all given non-free tuples in the subtree or r_i connects $c_{l,2}$, $c_{l,2}$ connects all given non-free tuples in the subtree or $\ldots\}$.

We use $CP(r_i, C_j)$ to denote the following probability: $Pr\{r_i$ connects $c_{j,1}$, $c_{j,1}$ connects all given non-free tuples in the subtree or r_i connects $c_{j,2}$, $c_{j,2}$ connects all given non-free tuples in the subtree or $\ldots\}$.

There are the following six cases:

1. C_j is corresponding to a leaf node and C_j is referred by the tuple set corresponding to the root ($R \rightarrow C_j$);
2. C_j is corresponding to a leaf node and the tuple set corresponding to the root is referred by C_j ($R \leftarrow C_j$);
3. C_j is corresponding to a single connection subtree root node and C_j is referred by the tuple set corresponding to the root ($R \rightarrow C_j$);
4. C_j is corresponding to a single connection subtree root node and the tuple set corresponding to the root is referred by C_j ($R \leftarrow C_j$);
5. C_j is corresponding to a multiple connection subtree root node and C_j is referred by the tuple set corresponding to the root ($R \rightarrow C_j$);
6. C_j is corresponding to a multiple connection subtree root node and the tuple set corresponding to the root is referred by C_j ($R \leftarrow C_j$).

For case (1), $CS(r_i, C_j) = 1/|C_j|$. For case (2), $CS(r_i, C_j) = 1/|R|$. For case (3), $CS(r_i, C_j) = CS(c_{j,k})$, where $c_{j,k}$ is a randomly chosen tuple in C_j. For case (4), $CS(r_i, C_j) = \frac{|C_j|}{|R|} \cdot CS(c_{j,k})$, where $c_{j,k}$ is a randomly chosen tuple in C_j. For case (5), $CS(r_i, C_j) = CS(c_{j,k})$, where $c_{j,k}$ is a randomly chosen tuple in C_j. For case (6), $CS(r_i, C_j) = 1 - (1 - \frac{1}{|R|} \cdot CS(c_{j,k}))^{|C_j|}$, where $c_{j,k}$ is a randomly chosen tuple in C_j.

B. THE JTT GENERATOR OPTIMIZER

We have the following observations:

1. For any two adjacent tuple sets $A \leftarrow B$ in the given CN, the corresponding join condition is $\bowtie_(aid = aid)$, where aid is the relation A's primary key.
2. The sizes of all non-free tuple sets are already known.
3. For any two adjacent tuple sets $A \leftarrow B$, each tuple $a \in A$ is referred by $\frac{size(B)}{size(A)}$ tuples in B on average.

Our objective is to retrieve a minimal number of tuples from DBMS. We have the following assumptions: (1) hash indices have been built on all primary keys and foreign keys; (2) we only use hash joins.

Since all non-free tuple sets are already given, we only need to retrieve extra tuples from free tuple sets. For each free tuple set A, we use S^A to denote the set of necessary tuples which should be retrieved from DBMS. Thus, each tuple in S^A must appear in at least one JTT. However, we do not know S^A in advance. We use R^A denote the set of

tuples we will retrieve. We must guarantee that $S^A \subseteq R^A$ (completeness) and we want to minimize $|R^A|$.

If there are n non-free tuple sets in the given CN, we divide all join and selection conditions into n parts. Each part corresponds to a path to a non-free tuple set. The following CN is an example: $A^Q \leftarrow B^{\{\}} \rightarrow C^Q$. We can divide join conditions into two parts, $R_1^B = \{b \in B^{\{\}} | \exists a \in A^Q \text{ s.t. } b.aid = a.aid\}$ and $R_2^B = \{b \in B^{\{\}} | \exists c \in C^Q \text{ s.t. } b.cid = c.cid\}$. R_1^B corresponds to a path to non-free tuple set A^Q and R_2^B corresponds to a path to non-free tuple set C^Q. Then, we know that $S^B = R_1^B \cap R_2^B$. So, $S^B \subseteq R_1^B$ and $S^B \subseteq R_2^B$. Therefore, R_1^B and R_2^B are two different choices of R^B. Note that $|A^Q|$ and $|C^Q|$ are known. Based on observation (3), $|R_1^B|$ and $|R_2^B|$ can be estimated easily along its corresponding path. For each free tuple set A, we can choose R_i^A with minimal estimation size as R^A. Then, we retrieve tuples in R^A along the corresponding path using type 1 and type 2 SQL queries.

Given a CN, an execution plan, and a block b, we want to enumerate all JTTs corresponding to the given CN in the block b. First, we retrieve each free tuple set separately using two types of simple SQL queries. As mentioned, we can divide the entire CN into n parts according to the chosen path to one of the n non-free tuple sets. Then, we join all parts together one by one. Each time we add a new part, check the join condition, and filter out all unnecessary tuples from each retrieved tuple sets. Finally, there is only one merged connected component left and all retrieved tuple sets only contain necessary tuples (a necessary tuple must appear in at least one JTT). We then enumerate JTTs from these tuple sets.