SPARK2: Top-k Keyword Query in Relational Databases

Yi Luo, Wei Wang, *Member, IEEE,* Xuemin Lin,
Xiaofang Zhou, *Senior Member, IEEE* Jianmin Wang, Keqiu Li

Abstract—**With the increasing amount of text data stored in relational databases, there is a demand for RDBMS to support keyword queries over text data. As a search result is often assembled from multiple relational tables, traditional IR-style ranking and query evaluation methods cannot be applied directly.**

In this paper, we study the *effectiveness* and the *efficiency* issues of answering top-k keyword query in relational database systems. We propose a new ranking formula by adapting existing IR techniques based on a natural notion of *virtual document*. We also propose several efficient query processing methods for the new ranking method. We have conducted extensive experiments on large-scale real databases using two popular RDBMSs. The experimental results demonstrate significant improvement to the alternative approaches in terms of retrieval effectiveness and efficiency.

Index Terms—**top-k, keyword search, relational database, information retrieval**

I. INTRODUCTION

Integration of DB and IR technologies has been an active research topic recently [1]. One fundamental driving force is the fact that more and more text data are now stored in relational databases. Examples include commercial applications such as customer relation management systems (CRM), and personal or social applications such as Web blogs and wiki sites. Since the dominant form of querying free text is through keyword search, there is a natural demand for relational databases to support *effective* and *efficient* IR-style keyword queries.

In this paper, we focus on the problem of supporting effective and efficient top-k keyword search in relational databases. While many RDBMSs support full-text search, they only allow retrieving relevant tuples from within the same relation. A unique feature of keyword search over RDBMSs is that search results are often *assembled* from relevant tuples in several relations such that they are *inter-connected* and *collectively* be relevant to the keyword query [2], [3]. Supporting such feature has a

Y. Luo is with Laboratory Le2i of CNRS Dijon, France.
E-mail: Yi.Luo@u-bourgogne.fr
W. Wang and X. Lin are with University of New South Wales, Australia.
E-mail: {weiw, lxue}@cse.unsw.edu.au
X. Zhou is with University of Queensland, Australia.
E-mail: zxf@itee.uq.edu.au
J. Wang is with Tsinghua University, China.
E-mail: jimwang@tsinghua.edu.cn
K. Li is with Dalian University of Technology, China.
E-mail: likeqiu@gmail.com

TABLE I
SEARCHING "2001 HANKS" ON IMDB.COM

1	2001: HAL's Legacy (2001) (TV)
2	Gigantic Skate Park Tour: Summer 2002 (2002) (TV)
3	TV Hunks and Babes 2006 (2006) (TV)

number of advantages. Firstly, data may have to be split and stored in different relations due to database normalization requirement. Such data will not be returned if keyword search is limited to only single relations. Secondly, it lowers the barrier for casual users to search databases, as it does not require users to have knowledge about query languages or database schema. Thirdly, it helps to reveal interesting or unexpected relationships among entities [4]. Lastly, for websites with database back-ends, it provides a more flexible search method than the existing solution that uses a fixed set of pre-built template queries. For example, we issued a search of "`2001 hanks`" using the search interface on imdb.com, and failed to find relevant answers (See Table I for the top-3 results returned). In contrast, the same search on our system (on a database populated with imdb.com's data) will return results shown in Table II, where relevant tuples from multiple relations (marked in bold font) are joined together to form a meaningful answer to the query.

TABLE II
TOP-3 SEARCH RESULTS ON OUR SYSTEM

1	**Movies**: "Primetime Glick" (2001) Tom Hanks/Ben Stiller (#2.1)
2	**Movies**: "Primetime Glick" (2001) Tom Hanks/Ben Stiller (#2.1) ← **ActorPlay**: Character = Himself → **Actors**: Hanks, Tom
3	**Actors**: John Hanks ← **ActorPlay**: Character = Alexander Kerst → **Movies**: Rosamunde Pilcher - Wind über dem Fluss (2001)

There has been many related work dedicated to keyword search in databases recently [2]–[10]. Among them, [7] first incorporates state-of-the-art IR ranking formula to address the retrieval effectiveness issue. It also presents several efficient query execution algorithms optimized for returning top-k relevant answers. The ranking formula is subsequently improved by Liu, *et al.* [10] by using several refined weighting schemes. BANKS [3] and BANKS2 [8] took another approach by modeling the database content as a graph and proposed sophisticated ranking and query execution algorithms. [4], [9] studied theoretical aspects of efficient query processing for top-k keyword queries.

Despite the existing studies, there are still several issues with existing ranking methods, some of which may even lead to search results contradictory to human perception. In this paper, we analyze shortcomings of previous approaches and propose a new ranking method by adapting existing IR ranking methods and principles to our problem based on a *virtual document* model. Our ranking method also takes into consideration other factors (e.g., completeness and size of a result). Another feature is the use of a single tuning parameter to inject AND or OR semantics into the ranking formula. The technical challenge with the new ranking method is that the final score of an answer is aggregated from multiple scores of each constituent tuples, yet the final score is *not* monotonic with respect to any of its sub-components. Existing work on top-k query optimization cannot be immediately applied as they all rely on the monotonicity of the score aggregation function. Therefore, we also study efficient query processing methods optimized for our non-monotonic scoring function. We propose a *skyline sweeping* algorithm that achieves minimal database probing by using a monotonic score upper bounding function for our ranking formula. We also explore the idea of employing another non-monotonic upper bounding function to further reduce unnecessary database accesses, which results in the *block pipeline* algorithm. We then propose the tree pipeline algorithm which share the intermediate results among CNs at a fine granularity. We have conducted extensive experiments on large-scale real databases on two popular RDBMSs. The experimental results demonstrate that our proposed approach is superior to the previous methods in terms of effectiveness and efficiency.

We summarize our contributions as:

- We propose a novel and nontrivial ranking method that adapts the state-of-the-art IR ranking methods to ranking heterogeneous joined results of database tuples. The new method addresses an important deficiency in the previous methods and results in substantial improvement of the quality of search results.
- We propose three algorithms, *skyline sweeping*, *block pipeline*, and *tree pipeline*, to provide efficient query processing mechanism based on our new ranking method. The key challenge is that the non-monotonic nature of our ranking function renders existing top-k query processing techniques inapplicable. Our new algorithms are based on several novel score upper bounding functions. They also have the desirable feature of interacting minimally with the databases.
- We conduct comprehensive experiments on large-scale real databases containing up to ten million tuples. Our experiment results demonstrated that the new ranking method outperformed alternative approaches, and that our query processing algorithms delivered superior performance to previous ones.

The rest of the paper is organized as follows: Section II provides an overview of the problem and existing solutions. Section III presents our new ranking method and Section IV introduces two query processing algorithms optimized for efficient top-k retrieval. Section V introduces a query processing algorithm that features fine granularity computational sharing between CNs. We introduce related work in Section VII and Section VIII concludes the paper.

II. PRELIMINARIES

A. Problem Overview and Problem Definition

We consider a relational schema $\mathcal{R}$ as a set of relations $\{R_1, R_2, \ldots, R_{|\mathcal{R}|}\}$. These relations are interconnected at the schema level via foreign key to primary key references. We denote $R_i \rightarrow R_j$ if R_i has a set of foreign key attribute(s) referencing R_j's primary key attribute(s), following the convention in drawing relational schema graphs. For simplicity, we assume all primary key and foreign key attributes are made of single attribute, and there is at most one foreign key to primary key relationship between any two relations. We do not impose such limitations in our implementation. A query Q consists of (1) a set of distinct keywords, i.e., $Q = \{w_1, w_2, \ldots, w_{|Q|}\}$; and (2) a parameter k indicating that a user is only interested in top-k results ranked by relevance scores associated with each result. Ties can be broken arbitrarily. A user can also specify AND or OR semantics for the query, which mandates that a result must or may not match *all* the keywords, respectively. The default mode is the OR semantics to allow more flexible result ranking [7].

A result of a top-k keyword query is a tree, T, of tuples, such that each leaf node of T contains at least one of the query keyword, and each pair of adjacent tuples in T is connected via a foreign key to primary key relationship. We call such an answer tree a *joined tuple tree* (**JTT**). The *size* of a JTT is the number of tuples (i.e., nodes) in the tree. Note that we allow two tuples in a JTT to belong to the same relation. Each JTT belongs to the results produced by a relational algebra expression — we just replace each tuple with its relation name and impose a full-text selection condition on the relation if the tuple is a leaf node. Such relational algebra expression (or its SQL equivalent) is also termed as *Candidate Network* (**CN**) [6]. Relations in the CN are also called *tuple sets*. There are two kinds of tuple sets: those that are constrained by keyword selection conditions are called *non-free tuple sets* (denoted as R^Q) and others are called *free tuple sets* (denoted as R). Every JTT as an answer to a query has its *relevance* score, which, intuitively, indicates how relevant the JTT is to the query. Conceptually, all JTTs of a query will be sorted according to the descending order of their scores and only those with top-k highest scores will be returned.

Example 2.1: In this paper, we use the same running example as the previous work [7] (shown in Figure 1).

In the example, $\mathcal{R} = \{P, C, U\}$.[1] Foreign key to primary key relationships are: $C \rightarrow P$ and $C \rightarrow U$. A user wants to retrieve top-3 answer to the query "maxtor netvista".

Some example JTTs include: c_3, $c_3 \rightarrow p_2$, $c_1 \rightarrow p_1$, $c_2 \rightarrow p_2$, and $c_2 \rightarrow p_2 \leftarrow c_3$. The first JTT belongs to CN

---

[1] Initials of relation names are used as shorthands (except that we use U to denote $Customers$).

Fig. 1. A Running Example from [7] (Query is "`maxtor netvista`"; Matches are Underlined)

C^Q; the next three JTTs belong to CN $C^Q \rightarrow P^Q$; and the last JTT belongs to CN $C^Q \rightarrow P^Q \leftarrow C^Q$. Note that $c_3 \rightarrow u_3$ is *not* a valid JTT to the query, as the leaf node u_3 does not contribute to a match to the query.

A possible answer for this top-3 query may be: c_3, $c_3 \rightarrow p_2$, and $c_1 \rightarrow p_1$. We believe that most users will prefer $c_1 \rightarrow p_1$ to $c_2 \rightarrow p_2$, because the former complaint is really about a IBM `Netvista` equipped with a `Maxtor` disk, and that it is not certain whether Product p_2 mentioned in the latter JTT is equipped with a `Maxtor` hard disk or not.

B. Overview of Existing Solutions

We will use the running example to briefly introduce the basic ideas of existing query processing and ranking methods.

Given the query keywords, it is easy to find relations that contain at least one tuple that matches at least one search keyword, if the system supports full-text query and inverted index. The matched tuples from those relations forms the *non-free tuple sets*, and are usually ordered in descending order by their IR-style relevance scores. The challenge is to find inter-connected tuples that collectively form valid JTTs. Given the schema of the database, we can *enumerate* all possible relational algebra expressions (i.e., CNs) such that each of them *might* generate an answer to the query.

Example 2.2: For the query "`maxtor netvista`", only P and C have tuples matching at least one keyword of the query. The non-free tuple set of C is $C^Q = [c_3, c_2, c_1]$, and the non-free tuple set of P^Q is $[p_1, p_2]$. The free tuple set of U is U itself. While $C^Q \rightarrow P^Q$ might produce an answer, $C^Q \rightarrow U$ cannot produce any valid answer (i.e., JTT), as the joining U tuple won't contribute any keyword match to the query. However, note that other larger CNs whose query expressions contain that of $C^Q \rightarrow U$ (e.g., $C^Q \rightarrow U \leftarrow C^Q$) may still produce an answer.

DISCOVER [6] has proposed a breadth-first CN enumeration algorithm that is both sound and complete. The algorithm is essentially enumerating all subgraphs of size k that does not violate any pruning rules. The algorithm varies k from 1 to some search range threshold M. Three pruning rules are used and they are listed below. We also show the traces of the CN generation algorithm running on our example (Table III).

Rule 1 : Prune duplicate CNs.

TABLE III
ENUMERATING CNs. P AND C BOTH MATCH THE TWO QUERY KEYWORDS. WE MARK INVALID CNs IN GRAY. (WE OMIT CNs PRUNED BY RULE 1)

Schema:		$P^Q \leftarrow C^Q \rightarrow U$		
Size	**CN ID**	**CN**	**Valid?**	**Violates**
1	CN_1	P^Q	**Y**	
1	CN_2	C^Q	**Y**	
2	CN_3	$P^Q \leftarrow C^Q$	**Y**	
2		$C^Q \rightarrow U$	n	Rule (2)
3		$P^Q \leftarrow C^Q \rightarrow U$	n	Rule (2)
3		$P^Q \leftarrow C^Q \rightarrow P^Q$	n	Rule (3)
3	CN_4	$C^Q \rightarrow P^Q \leftarrow C^Q$	**Y**	
3		$U \leftarrow C^Q \rightarrow U$	n	Rules (2, 3)
3	CN_5	$C^Q \rightarrow P \leftarrow C^Q$	**Y**	
3	CN_6	$C^Q \rightarrow U \leftarrow C^Q$	**Y**	
4	$\vdots$	$\vdots$	$\vdots$	

Rule 2 : Prune non-minimal CNs, i.e., CNs containing at least one leave node which does not contain a query keyword.

Rule 3 : Prune CNs of type: $R^Q \leftarrow S^* \rightarrow R^Q$. The rationale is that any tuple $s \in S^*$ (S^* may be a free or non-free tuple set) which has a foreign key pointing to a tuple in R^Q must point to the same tuple in R^Q.

Four valid CNs (CN_1 to CN_4) are found in the above example. Each CN naturally corresponds to a database query. E.g., CN_3 corresponds to the following SQL statement in Oracle's syntax:

```
SELECT *
FROM    Products P, Complaints C
WHERE   P.prodId = C.prodId
  AND   (CONTAINS(P.manufacturer,
                  'maxtor, netvista') > 0
   OR    CONTAINS(P.model,
                  'maxtor, netvista') > 0)
  AND   CONTAINS(C.comments ,
                  'maxtor, netvista') > 0
```

To find top-k answers to the query, a naïve solution is to issue an SQL query for each CN and union them to find the top-k results by their relevance scores. DISCOVER2 [7] introduce two alternative query evaluation strategies: *sparse* and *global pipeline* algorithms, both optimized for stopping the query execution immediately after the true top-k-th

result can be determined.[2] The basic idea is to use an upper bounding function to bound the scores of potential answers from each CN (either before execution or in the middle of its execution). The upper bound score ensures that any potential result from future execution of a CN will *not* have a higher score. Thus the algorithm can stop earlier if the current top-k-th result has a score no smaller than the upper bound scores of all CNs. We note that this is the main optimization technique for other variants of top-k queries too [11]–[13].

The sparse algorithm executes one CN at a time and updates the current top-k results; it uses the above-mentioned criterion to stop query execution earlier. The global pipeline algorithm adopts a more aggressive optimization: it does not execute a CN to its full; instead, at each iteration, it (a) first selects the most *promising* CN, i.e., the CN with the highest upper bound score; (b) admits the next unseen tuple from one of the CN's non-free tuple sets and join the new tuple with *all* the already seen tuples in all the other non-free tuple sets. As such, the query processing strategy (of a single CN) is similar to that of *ripple join* [14].

C. Overview of Our Solution

In this paper, we assume that the DBMS can efficiently locate the matching tuples for each search keyword and form the non-free tuple sets. We will focus on the following two sub-problems: (a) how to score a JTT, and (b) how to generate and order the SQL queries for the CNs of a query, such that minimal database accesses (also called *probes*) are required before top-k results are returned

The first problem is studied in the next section. The second problem is addressed in Section IV.

III. RANKING FUNCTION

Due to the fuzzy nature of keyword queries, retrieval effectiveness is vital to keyword search on RDBMSs. The initial attempt was a simple ranking by the size of CNs [2], [6]. DISCOVER2 later proposed a ranking formula based on the state-of-the-art IR scoring function [7]. More recently, several sophisticated improvements to the ranking formula in [7] have been suggested [10].

In this section, we first motivate our work by presenting observations that reveal several problems in the existing schemes. We show that simply aggregating the the IR scores for each individual tuple in a JTT violates the IR scoring principle and results in anomalies. We then propose to model a JTT as a virtual document by combining all its tuples together and then computing a holistic score. Our final scoring function also takes into consideration the number of query keyword matched in a JTT and the size of the JTT.

A. Problems with Existing Ranking Functions

The basic idea of the ranking method used in DIS-COVER2 [7] (and its variant [10]) is to

1) assign each tuple in the JTT a score using a standard IR-ranking formula (or its variants); and
2) combine the individual scores together using a score aggregation function, $comb(\cdot)$, to obtain the final score. Only monotonic aggregation functions, e.g., SUM, have been considered.[3]

For example, the IR-style ranking function used in DIS-COVER2 is adapted from the TF-IDF ranking formula as:[4]

$$score(T, Q) = \sum_{t \in T} score(t, Q)$$

$$score(t, Q) = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(tf_w(t)))}{(1 - s) + s \cdot \frac{dl_t}{avdl_t}} \cdot \ln(idf_w)$$

where $idf_w = \frac{N_{Rel(t)} + 1}{df_w(Rel(t))}$

$tf_w(t)$ denotes the number of times a keyword w appears in a database tuple t, dl_t denotes the length of the text attribute of a tuple t, and $avdl_t$ is the average length of the text attribute in the relation which t belongs to (i.e., $Rel(t)$), $N_{Rel(t)}$ denotes the number of tuples in $Rel(t)$, and $df_w(Rel(t))$ denotes the number of tuples in $Rel(t)$ that contain keyword w. The score of a JTT is the sum of the *local* scores of every tuple in the JTT.

TABLE IV
DIFFERENT SCORING FUNCTIONS PRODUCES DIFFERENT RANKINGS
($\ln(idf_{\texttt{maxtor}}) = \ln(idf_{\texttt{netvista}}) = 1.0$ AND $dl_t = avdl_t$)

CN	$t \in CN$	$tf_{\texttt{maxtor}}$	$tf_{\texttt{netvista}}$	$Score_t$	$Score_T$	Our Score
$c_3 \to p_2$	c_3	1	1	2.0	3.0	1.13
	p_2	0	1	1.0		
$c_1 \to p_1$	c_1	0	1	1.0	2.0	0.98
	p_1	1	0	1.0		
$c_2 \to p_2$	c_2	0	1	1.0	2.0	0.44
	p_2	0	1	1.0		

We illustrate an inherent problem in the above framework by using the running example in Figure 1. The query is "maxtor netvista". Let us consider the CN: $C^Q \to P^Q$. If the CN is executed completely, it will produce 3 results. In Table IV, we list the detailed steps to obtain the scores ($Score_T$) according to the above-mentioned method. For example, $c_3 \to p_2$ consists of two tuples c_3 and p_2 belonging to C and P, respectively. c_3 contains *one* maxtor and *one* netvista, while p_2 contains *one* netvista only. For simplicity, we do not consider length normalization in the example (i.e., setting $dl_t = avdl_t$ for all t), and assume that the $\ln(idf)$ values of both keywords are 1. Therefore, we can calculate $score(c_3, Q)$ as $1+\ln(1+\ln(tf_{\texttt{maxtor}}(c_3))+1+\ln(1+\ln(tf_{\texttt{netvista}}(c_3)) =$

2.0, and $score(p_2, Q) = 1 + \ln(1 + \ln(tf_{\texttt{netvista}}(p_2))) = 1.0$. The final score for the joined tuple, $score(c_3 \rightarrow p_2)$, is $2.0 + 1.0 = 3.0$. Similarly, $c_1 \rightarrow p_1$ and $c_2 \rightarrow p_2$ both have the same score 2.0 and thus are both ranked as the second.

However, a careful inspection of the latter two results reveals that $c_1 \rightarrow p_1$ in fact matches *both* search keywords while $c_2 \rightarrow p_2$ matches only one keyword (`netvista`) albeit twice. We believe that most users will find the former answer more relevant to the query than the latter one. In fact, it is not hard to construct an extreme example where the DISCOVER2's ranking contradicts human perception by ranking results that contain a large amount of one search keyword over results that contain all or most search keywords but only once.

There are two reasons for the above-mentioned ranking problem. Firstly, when a user inputs short queries, there is a strong implicit tendency for the user to prefer answers matching queries completely to those matching queries partially. We propose a *completeness factor* in Section III-C to quantify this factor. Secondly, the framework of combining local IR ranking scores has an inherent side effect of overly rewarding contributions of the *same* keyword in different tuples in the same JTT.

We note that a similar observation and remedy about the need of non-linear term frequency attenuation was also made by IR researchers [15]. The difference is that the same approach is motivated by the semantics of our search problem; in addition, our problem is more general and a number of other modifications to the IR ranking function are made (e.g., inverse document frequencies and document length normalization for each CN).

B. Modelling a Joined Tuple Tree as a Virtual Document

We propose a solution based on the idea of modelling a JTT as a *virtual document*. Consequently, the entire results produced by a CN will be modeled as a document collection. The rationale is that most of the CNs carry certain distinct semantics. E.g., $C^Q \rightarrow P^Q$ gives *all* details about complaints and their related products that are collectively relevant to the query Q and form integral *logical* information units. In fact, the actual lodgment of a complaint would contain both product information and the detailed comment — it was split into multiple tables due to the normalization requirement imposed by the *physical* implementation of the RDBMSs.

A very similar notion of *virtual document* was proposed in [16]. Our definition differs from [16] in that ours is query-specific and dynamic. For example, a customer tuple is only joined with complains matching the query to form a virtual document on the run-time, rather than joining with *all* the complaints as [16] does.

By adopting such a model, we could naturally compute the IR-style relevance score without using an esoteric score aggregation function. More specifically, we assign an IR ranking score to a JTT T as

$$score_a(T, Q) = \sum_{w \in T \cap Q} \frac{1 + \ln(1 + \ln(tf_w(T)))}{(1 - s) + s \cdot \frac{dl_T}{avdl_{CN^*(T)}}} \cdot \ln(idf_w) \tag{1}$$

where $tf_w(T) = \sum_{t \in T} tf_w(t), \quad idf_w = \frac{N_{CN^*(T)} + 1}{df_w(CN^*(T))}$

$CN(T)$ denotes the CN which the JTT T belongs to, $CN^*(T)$ is identical to $CN(T)$ except that all full-text selection conditions are removed. $CN^*(T)$ is also written as CN^* if there is no ambiguity.

Example 3.1: Consider the CN $C^Q \rightarrow P^Q$, CN^* is $C \rightarrow P$ (i.e., $C \bowtie P$) in Table IV. $N_{CN^*} = 3$. $df_{\texttt{maxtor}} = 2$ and $df_{\texttt{netvista}} = 3$.

In our proposed method, the contributions of the same keyword in different relations are *first* combined and *then* attenuated by the term frequency normalization. Therefore, $tf_{\texttt{maxtor}}(c_2 \rightarrow p_2) = 0$, $tf_{\texttt{netvista}}(c_2 \rightarrow p_2) = 2$, while $tf_{\texttt{maxtor}}(c_1 \rightarrow p_1) = 1$, $tf_{\texttt{netvista}}(c_1 \rightarrow p_1) = 1$. According to Equation (1) and omitting the size normalization, $score_a(c_2 \rightarrow p_2) = 0.44$, while $score_a(c_1 \rightarrow p_1) = 0.98$. Thus, $c_1 \rightarrow p_1$ is ranked higher than $c_2 \rightarrow p_2$, which agrees with human judgments.[5]

There are still two technical issues remaining: how to obtain $df_w(CN^*)$ and N_{CN^*} and how to obtain $avdl_{CN^*}$. No doubt that computing $df_w(CN*)$ and N_{CN*} exactly will incur prohibitive cost. One solution is to compute them approximately: we estimate $p = \frac{df_w(CN^*)}{N_{CN^*}}$, such that the idf value of the term in CN^* can be approximated as $\frac{1}{p}$. Consider a $CN^* = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_l$, and denote the percentage of tuples in R_j that matches at least a keyword w as $p_w(R_j)$. We can derive

$$\frac{df_w(CN^*)}{N_{CN^*} + 1} \approx \frac{df_w(CN^*)}{N_{CN^*}} = p \approx 1 - \Pi_j(1 - p_w(R_j))$$

by assuming that (a) N_{CN*} is a large number, and (b) tuples matching keyword w are uniformly and independently distributed in each relation R_j. In a similar fashion, we estimate $avdl_{CN^*}$ as $\sum_j avdl_{R_j}$.

C. Other Ranking Factors

Completeness Factor: As motivated in Section III-A, we believe that users usually prefer documents matching many query keywords to those matching only few keywords. To quantify this factor, we propose to multiply a *completeness* factor to the raw IR ranking score. We note that the same intuition has been recognized by IR researchers when studying ranking for *short queries* [17], [18].

Our proposed completeness factor is derived from the *extended Boolean model* [19]. The central idea of the extended Boolean model is to map each document into a point in a m-dimensional space $[0, 1]^m$, if there are m keywords in the query Q. A document d will have a large

---

[5]$c_3 \leftarrow p_2$ will have score 1.13, which still makes it ranked as the first result.

coordinate value on a dimension, if it has high relevance to the corresponding keyword. As we prefer documents containing all the keywords, the *ideal* answer should be located at the position $P_{ideal} = [\underbrace{1, \ldots, 1}_{m}]$. In our virtual document model, a JTT is a document and can be projected into this m-dimensional space just as a normal document. We thus use the distance of a document to the ideal position, P_{ideal}, as the *completeness value* of the JTT. More specifically, we use the L_p distance and normalize the value into $[0, 1]$. The completeness factor, $score_b$, is then defined as:

$$score_b(T, Q) = 1 - \left(\frac{\sum_{1 \leq i \leq m} (1 - T.i)^p}{m} \right)^{\frac{1}{p}} \quad (2)$$

where $T.i$ denotes the normalized term frequency of a JTT T with respect to keyword w_i, i.e.,

$$T.i = \frac{tf_{w_i}(T)}{\max_{1 \leq j \leq m} tf_{w_j}(T)} \cdot \frac{idf_{w_i}}{\max_{1 \leq j \leq m} idf_{w_j}}$$

In Equation (2), p is a tuning parameter. p can smoothly switch the completeness factor biased towards the OR semantics to the AND semantics, when p increases from 1.0 to ∞. To see that, consider $p \to \infty$, the completeness factor will essentially become $\min_{1 \leq i \leq m} T.i$, which essentially gives 0 score to a result failing to match all the search keywords. In our experiment, we observed that a p value of 2.0 is already good enough to enforce the AND-semantics for almost all the queries tested.

Apart from the nice theoretical properties, the ability to switch between AND and OR semantics is a salient feature to query processing. It enables a unified framework optimized for top-k query processing for both AND and OR semantics. In contrast, previous approaches are either optimized for the AND semantics [2] or for the OR semantics [7].

Size Normalization Factor: The size of the CN or JTT is also an important factor. A larger JTT tends to have more occurrences of keywords. A straightforward normalization by $\frac{1}{size(CN)}$ [7] usually penalizes too much for even moderate-sized CNs. We experimentally found that the following size normalization factor works well in the experiment:

$$\begin{aligned} score_c = &(1 + s_1 - s_1 \cdot size(CN)) \\ &\cdot (1 + s_2 - s_2 \cdot size(CN^{nf})) \end{aligned} \quad (3)$$

where $size(CN^{nf})$ is the number of non-free tuple sets for the CN. In our experiments, we found that $s_1 = 0.15$ and $s_2 = \frac{1}{|Q|+1}$ yielded good retrieval results for most of the queries.

D. The Final Scoring Function

In summary, our ranking method can be conceptually thought as first merging all the tuples in a JTT into a virtual document, and then obtaining its IR ranking score (Equation (1)), the completeness factor score (Equation (2)), and the size normalization factor score (Equation (3)). Since each component score reflects the quality of the answer in different perspectives, we obtain the final score of the JTT, $score(T, Q)$, as the product of all the three scores:

$$score(T, Q) = score_a(T, Q) \cdot score_b(T, Q) \cdot score_c(T, Q)$$

IV. TWO TOP-k JOIN ALGORITHMS

While effectiveness of keyword search is certainly the most important factor, we believe that the efficiency of query processing is also a critical issue. Query execution time will become prohibitively large for large databases, if the query processing algorithm is not fully optimized for the ranking function and top-k queries.

In this section, we propose two efficient query processing algorithms for our newly proposed ranking function. The first algorithm carefully constructs a minimal group of potential solutions by observing the score dominance relationship between candidates solutions. This results in the Skyline Sweeping algorithm which is optimal in terms of number of database probes. The second algorithm partitions the tuples in a CN into blocks according to their signatures and can effectively alleviate the inefficiency inherent in our complex, non-monotonic scoring function. Yet another algorithm that further exploits computational sharing between CNs will be given in Section V.

A. Dealing with Non-monotonic Scoring Function

The technical challenge of query processing mainly lies with the non-monotonic scoring function (mainly the $score_a(\cdot)$ and $score_b(\cdot)$ functions) used in our ranking method. To the best of our knowledge, none of the existing top-k query processing methods deals with non-monotonic scoring function. We use the single pipeline algorithm [7] to illustrate the challenge and motivate our algorithms.
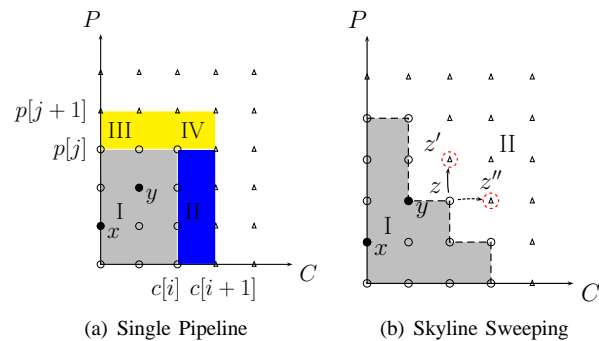

(a) Single Pipeline   (b) Skyline Sweeping

Fig. 2.   Query Evaluation Strategies

Example 4.1: Figure 2(a) illustrates a snapshot of running the single pipeline algorithm on the CN $C \to P$. Assume that we have processed the light gray area marked with "I" (i.e., the rectangle up to $(c[i], p[j])$). We use the notation $c[i]$ to denote the i-th tuple in the non-free tuple set C^Q in descending order of their scores.

In the figure, hollow circles denote *candidates* that we have examined but did not produce any result, filled circles

denote joined results, and hollow triangles denote candidates that have not been examined.

Assume that a user asks for top-2 results, and we have already found two results x and y. The single pipeline algorithm needs to decide whether to continue the query executing (and check more candidates) or stop and return $\{x, y\}$ as the query result. DISCOVER2 needs to bound the maximum score that any unseen candidate can achieve. If the last seen candidate is $c[i] \to p[j]$, then the upper bound is $\max(score(p[1], c[i+1]), score(p[j+1], c[1])$. This is true because DISCOVER2 uses a monotonic scoring function (SUM) and therefore $score(p[1], c[i+1]) \geq score(p[u], c[v])$ ($u \geq 1, v \geq i + 1$) and $score(p[j + 1], c[1]) \geq score(p[u], c[v])$ ($u \geq j + 1, v \geq 1$); the combination of the right-hand sides in the above two inequalities covers all the unseen candidates (i.e., those marked as triangles).

We note that, with our new ranking method, the score of a JTT is *not* monotonic with respect to the score of its constituent tuples. For example, consider the last two JTTs in Table IV. If we assume $idf_{\texttt{netvista}} > idf_{\texttt{maxtor}}$, then $score(c_2) = score(c_1)$ but $score(p_2) > score(p_1)$. However, we have $score(c_2 \to p_2) < score(c_1 \to p_1)$, even if we do not impose the penalty from the completeness factor. Consequently, previous algorithms on top-k query processing cannot be immediately applied, and a naïve approach would have to produce all the results to find the top-k results.

Our solution, which underlies both of our proposed algorithms, is based on the observation that if we can find a (preferably tight) *monotonic*, *upper bounding* function to the actual scoring function, we can stop the query processing earlier too. We derive such an upper bounding function for our ranking function in the following.

Let us denote a JTT as T, which consists of tuples $t_1, \ldots, t_m$. Without loss of generality, we assume every t_i is from a non-free tuple set; otherwise, we just ignore it from the subsequent formulas. Let $sumidf = \sum_{w \in CN(T) \cap Q} idf_w$ and $watf(t_i) = \frac{\sum_{w \in t_i \cap Q} (tf_w(t_i) \cdot idf_w)}{sumidf}$ (i.e., pseudo weighted average tf of tuple t_i). Then we have the following lemma. [6]

Lemma 4.1: $score_a(T, Q)$ (Equation (1)) can be bounded by a function $uscore_a(T, Q) = \frac{1}{1-s} \cdot \min(A, B)$, where

$$A = sumidf \cdot \left(1 + \ln\left(1 + \ln\left(\sum_{t_i \in T \cap Q} watf(t_i)\right)\right)\right)$$
$$B = sumidf \cdot \sum_{t_i \in T \cap Q} watf(t_i) .$$

In addition, the bound is tight.

A tight upper bound for the completeness factor (denoted as $uscore_b$) can be determined given the keywords matched in each non-free tuple sets of a CN. The size normalization factor is also a constant for a given CN. Therefore, we have the following theorem to upper bound the score of a JTT.

---

[6] Since idf_w is monotonically decreasing with the size of CNs, we can use the maximum idf_w value for all size-1 CNs here.

Theorem 4.1:

$$score(T, Q) \leq uscore(T, Q), \quad \text{where} \qquad (4)$$
$$uscore(T, Q) = uscore_a(T, Q) \cdot uscore_b(CN(T), Q)$$
$$\cdot score_c(CN(T), Q)$$

and for a given CN, the upper bound score is *monotonic* with respect to $watf(t_i)$ ($t_i \in T$).

This result immediately suggests that we should sort all the tuples (t_i) in the non-free tuple set of a CN by the decreasing order of their $watf(t_i)$ values (rather than their local IR scores as used in previous work), such that we can obtain an upper bound score of all the unseen candidates.

Example 4.2: Continuing the previous example, assume that we have ordered all tuples t in C^Q and P^Q according to the descending order of their $watf(t)$ values. Then the score of the unseen candidates in the CN: $X = C^Q \to P^Q$ is bounded by $M \cdot uscore_b(X, Q) \cdot score_c(X, Q)$, where $M = \max(uscore_a(c[i+1], p[1]), uscore_a(c[1], p[j+1]))$.

B. Skyline Sweeping Algorithm

Based on Theorem 4.1, we could modify the existing single or global pipeline algorithm such that it will correctly compute the top-k answers for our new ranking function. However, single/global pipeline algorithm may incur many *unnecessary* join checking. Therefore, we design a new algorithm, *skyline sweeping*, that is guaranteed *not* to incur any unnecessary checking and thus has the minimal number of accesses to the database.

Example 4.3: Consider the single pipeline algorithm running on the example in Figure 2(a). Assume that the algorithm has processed $c[1] \ldots c[i]$ on non-free tuple set C^Q and $p[1] \ldots p[j]$ on P^Q. If the algorithm cannot stop, it will pick up either $c[i+1]$ or $p[j+1]$. If it picks $c[i+1]$, j probing queries will be sent to verify whether $c[i + 1]$ joins with $p[k]$, where $1 \leq k \leq j$.

It is obvious that some of these j queries might be unnecessary, if, e.g., if $c[i + 1]$ joins with $p[1]$ and its real score is higher than the upper bound scores of the rest of the candidates, then the other $j - 1$ probes will be unnecessary.

We propose an algorithm designed to minimize the number of join checking operations, which typically dominates the cost of the algorithm. Our intuition is that if there are two candidates x and y and the upper bound score of x is higher than that of y, y should *not* be checked unless x has been checked. Therefore, we should arrange all the candidates to be checked according to their upper bound scores. A naïve strategy is to calculate the upper bound scores for all the candidates, sort them according to the upper bound scores, and check them one by one according to this optimal order. This will incur excessive amount of unnecessary work, since not all the candidates need to be checked.

We take the following approach. We define a *dominance* relationship among candidates. Denote $x.d_i$ as the order (i.e., according to their $watf$ values) of candidate x on the non-free tuple set d_i. If $x.d_i \leq y.d_i$ for all non-free tuple set d_i, then $uscore(x) \leq uscore(y)$. This enables us to

compute the upper bound score and check candidates in a *lazy* fashion: immediately after we check a candidate x, we push all the other candidates directly dominated by x into a priority queue by the descending order of their upper bound scores. It can be shown that the candidates in the queue form a *skyline* [20] and the skyline sweeps across the Cartesian space of the CN as the algorithm progresses, hence the name of the algorithm.

Algorithm 1: Skyline Sweeping Algorithm

1 Q.push($(\overbrace{1, 1, \ldots, 1}^{m})$, calc_uscore($(\overbrace{1, 1, \ldots, 1}^{m})$));
2 top-k $\leftarrow \emptyset$;
3 **while** top-k$[k].score < Q$.head()$.uscore$ **do**
4      $head \leftarrow Q$.pop_max();
5      $r \leftarrow$ executeSQL(formQuery($head$));
6   **if** $r \neq$ **nil then**
7   top-k.push(r, score(r));
8   **for** $i \leftarrow 1$ **to** m **do**
9          $t \leftarrow head$.dup();
10         $t.i \leftarrow t.i + 1$;
11   Q.push(t, calc_uscore(t)); /* According to Equation (4) */;
12   **if** $head.i > 1$ **then**
13   **break**;

14 **return** top-k;

The algorithm is shown in Algorithm 1. A result list, *top-k*, contains no more than k results ordered by the descending real scores. The main data structure is a priority queue, Q, containing all the candidates (which are mapped to multi-dimensional points) according to the descending order of their upper bound scores. The algorithm also maintains the invariant that the candidate at the head of the priority queue has the highest upper bound score among all candidates in the CN. The invariant is maintained by (a) pushing the candidate formed by the top tuple from all dimensions into the queue (Line 1), and (b) whenever a candidate is popped from the queue, its *adjacent* candidates are pushed into the queue together with their upper bounds (Lines 8–13). The algorithm stops when the real score of the current top-k-th result is no smaller than the upper bound score of the head element of the priority queue; the latter is exactly the upper bound score of all the unprocessed candidates.

A technical point is that we should avoid inserting the same candidate multiple times into the queue. Doing duplicate checking is inefficient in terms of time and space. We adopt a space partitioning method to totally avoid generating duplicate candidates. This is implemented in Lines 12–13 using the same ideas as [12]. For example, in Figure 2(b), assume the order of the dimensions is P, C. Both z' and z'' are the adjacent candidates to z, but only z' will be pushed into Q when z is examined by the algorithm.

Theorem 4.2: The skyline sweeping algorithm has the minimal number of probing to the database.

Generalizing to Multiple CNs: The skyline sweeping algorithm can be easily generalized to support more than one CN. The only modification is to change the initialization step: we just push the top candidate of each CN to the priority queue Q.

C. Block Pipeline Algorithm

We present another algorithm to further improve the performance of the skyline sweeping algorithm. We observe that the aggregation function we used is non-monotonic, yet, in order to stop execution earlier, we *have to* use a monotonic upper bounding function to bound it. As such, the upper bounding may be rather loose at places.

Large gaps between the upper bound scores and the corresponding real scores cause two problems in the skyline sweeping algorithm: (a) it is harder to stop the execution, as the upper bound of unprocessed candidates may be much higher than their real score, and consequently higher than the real score of the top-k-th result, and (b) the order of the probes is not optimal, as the algorithm will perform large number of probes and only obtain candidates with rather low real score, which cannot contribute to the final top-k answer.

In order to address the above problems, we propose a novel *block pipeline* algorithm. The central idea of the algorithm is to employ another *local non-monotonic* upper bounding function that bounds the real score of JTTs more accurately. As such, we will check the most promising candidates first and thus further reduce the number of probes to the database.

To illustrate the idea, we define several concepts first. Consider a non-free tuple set R^Q and a query $Q = \{w_1, \ldots, w_m\}$. We define the *signature* of a tuple t in R^Q as an ordered sequence of term frequencies for all the query keywords, i.e., $\langle tf_{w_1}(t), \ldots, tf_{w_m}(t) \rangle$. Then, we can partition each R^Q into a number of *strata* such that all tuples within the same stratum have the same signature (also called *the signature of the stratum*). For a given CN, the partitioning of its non-free tuple sets naturally induces a partitioning of all the join candidates. We call each partition of the join candidates a *block*. The signatures of the strata that forms a block b can be summed up as $\langle \sum_{t_i \in T} tf_{w_1}(t_i), \ldots, \sum_{t_i \in T} tf_{w_m}(t_i) \rangle$, to form the signature of the block (denoted as $sig(b)$).

If two candidates in the same block both pass the join test, they should have *similar* real scores, as they agree on the term frequencies of all the query keywords (and thus the completeness wrt. the query), and the size of the result. This observation helps to derive a much tighter upper bounding function, $bscore$, for any candidate T within the same block via the block signature:

$$bscore(b, Q) = \sum_{w \in Q \cap b} \frac{1 + \ln(1 + \ln(sig_w(b)))}{1 - s} \cdot \ln(idf_w)$$
$$\cdot \; score_b(b, Q) \cdot score_c(CN(T), Q) \quad (5)$$

We note that this new bounding function, albeit being tighter (as it is no larger than $uscore(T, Q)$ defined in

Lemma 4.1), *cannot* be directly used to derive the stopping condition for top-k query processing algorithms, as it is not monotonic with respect to any single computable measure of its non-free tuple sets.

Algorithm 2: Block Pipeline Algorithm

Input : $\mathcal{CN}$ is the set of CNs
1 $Q \leftarrow \emptyset$;
2 **forall** $cn \in \mathcal{CN}$ **do**
3      $b \leftarrow$ the first block of cn;
4      $b.status \leftarrow$ USCORE;
5   Q.push(b, calc_uscore(b));   /* According to Equation (4) */;
6 **while** $top\text{-}k[k].score < Q$.head().getScore() **do**
7      $head \leftarrow Q$.pop_max();
8   **if** $head.status =$ USCORE **then**
9          $head.status \leftarrow$ BSCORE;
10   Q.push($head$, calc_bscore($head$));   /* based on Eq. (5) */;
11   **forall** the adjacent blocks b' to $head$ enumerated in a non-redundant way **do**
12             $b'.status \leftarrow$ USCORE;
13   Q.push(b', calc_uscore(b'));
14   **else if** $head.status =$ BSCORE **then**
15         $R \leftarrow$ executeSQL(formQuery(b));
16   **forall** result $t \in R$ **do**
17             $t.status \leftarrow$ SCORE;
18   Q.push(r, calc_score($head$)); /* compute the real score */;
19   **else**
20   Insert $head$ into $top\text{-}k$;
21 **return** top-k;

We introduce a solution using lazy block calculation and integrate it with the monotonic upper bounding score function (Equation (4)) seamlessly. Algorithm 2 describes the pseudo-code of the *block pipeline* algorithm. Intuitively, the algorithm is "unwilling" to issue a database probing query if the current top-ranked item in the priority queue is only associated with its upper bound score ($uscore$), as the score might not be close enough to its real score. Our non-monotonic bounding function plays its role here by re-inserting the item back to the priority queue, but with its $bscore$ (Lines 9–10).

Theorem 4.3: The block pipeline algorithm will never be worse than the skyline sweeping algorithm in terms of number of probes to the database. When the score aggregation function is non-monotonic, there exists a database instance such that the block pipeline algorithm will check fewer candidates than the skyline sweeping algorithm.

Example 4.4: Consider the example in Figure 2(a) and assume that we have only $i+1$ tuples in the non-free tuple set C^Q and $j+1$ tuples in P^Q. Further assume that both $c[1], \ldots, c[i]$ and $p[1] \ldots, p[j]$ are tuples matching the same keyword, w_1, once (and thus form two strata), and both $c[i+1]$ and $p[j+1]$ match w_2 once (and form another two strata). Assume the idf values of w_1 is higher than that of w_2, and hence the strata containing matches of w_1 is ranked above those matching w_2. This gives us four blocks. E.g., block I is $[c[1] \ldots c[i]] \times [p[1] \ldots p[j]]$, and its block

signature is $\langle 2, 0 \rangle$. Similarly, block II and III have the same block signature as $\langle 1, 1 \rangle$.

We assume $\ln(idf_{w_1}) = 1.1$, $\ln(idf_{w_2}) = 1.0$, the completeness factor, $score_b$, is 0.5, the size normalization factor, $score_c$, is 1.0, and s is 0.2. We calculate the $bscore$s and $uscore$s for each block in the following table:

Block	$bscore$	$uscore$
I	1.05	2.74
II	2.63	2.63
III	2.63	2.63
IV	0.95	2.50

The skyline sweeping algorithm will inspect tuples in Block I first, then Block II and III, as tuples in Block I all have higher $uscore$s than those in Block II or III. However, all answers in Block I, if any, will have rather low scores (no higher than 1.05), and are not likely to become top-k results.

In contrast, in the block pipeline algorithm, even though Block I is pushed into the queue first (Lines 3–5), it is re-inserted with its $bscore$ (calculated by calc_bscore) as 1.05. Blocks II and III will go through the same process, but they will both be associated with a $bscore$ of 2.63. Thus they will both be checked against the database before any candidate in Block I. Furthermore, if k results are found after evaluating candidates in Blocks II and III and the real score of the top-k-th result is higher than 1.05, the block pipeline algorithm can terminate immediately.

D. Discussion

Instance optimality is a notion proposed by Fagin *et al.* [11] to assert that the cost of an algorithm is bounded by a constant factor of any other correct algorithms on all database instances. This notation is widely studied and adopted in most top-k query processing work.

We note that although the skyline sweeping algorithm can be shown to be instance-optimal, this notion of optimality is not helpful in our problem setting. Consider a single CN. If the skyline sweeping algorithm accesses d_i tuples from the each of the m non-free tuple sets, and let $d = \max_{1 \le i \le m} d_i$, we can show that any other algorithm must at least access at least d tuples. Therefore, the total cost in terms of tuple accesses of the skyline sweep algorithm can be bounded by an m-factor of other algorithms. However, the dominant cost in our problem setting is the cost of probing the database. For a large CN with a number of free and non-free tuple sets, each probe is a complex query involving joins of multiple relations. In contrast, sequentially accessing tuples in the non-free tuple sets is practically an inexpensive in-memory operation. Consider the sketch of proof of the instance-optimality above, it is possible that the skyline sweeping algorithm has to probe the database $O(d^m)$ times, while another algorithm only needs to probe the database for $O(d)$ times. As a result, the cost ratio cannot be bounded by a constant factor.

V. TREE PIPELINE ALGORITHM

It is observed that there are substantial sharing of common join expressions among CNs. Hence substantial

computational efforts can be saved if multiple CNs can be executed/probed in a calculated way that maximizes the sharing of intermediate results; in addition, an empty intermediate result is useful in pruning any other CN that contains it.

Example 5.1: Consider a particular block b in $CN_4 = C^Q \rightarrow P^Q \leftarrow C^Q$ with tuple signatures α, β, and γ on the non-free tuple set C^Q, P^Q, and C^Q, respectively. Now consider $CN_3 = P^Q \leftarrow C^Q$. It comprises of a block b' which has the same signature for every non-free tuple set as b, e.g., b' also has signature of α on C^Q (b' is called a projection of b). We have the following observations:

1) *Pruning.* If b' has been executed and the result is empty, we know b will have empty result. In fact, any other block that has b as the projection will have empty result too.

2) *Reusing Partial Results.* If b' has been executed and the result is not empty and has been *cached*, we can use the result for executing block b rather than starting from scratch.

In the rest of this section, we propose the Tree Pipeline Algorithm (TPA). We first introduce the notation of the *partitioning graph*, which captures the sharing relationships among CNs. We then propose query processing algorithms that efficiently utilizes the intermediate results.

A. Partitioning Graph

We introduce the concept of *partition graph* which captures *all* the binary *decompositions* of CNs.

Assume that CN_1 and CN_2 are two valid CNs. We say that CN_1 is a *sub-CN* of CN_2 if the graph representation of CN_1 is a sub-graph of that of CN_2; or we can say CN_2 *contains* CN_1. We use $b.CN$ to denote the CN the block b belongs to. Given a CN CN_2, its sub-CN CN_1 and a block b in CN_2, the *projection* of b on CN_1 is a block b' in CN_1 such that on each common non-free tuple set of CN_1 and CN_2, the signatures of b' and b are the same.

Next, we define a *binary decomposition* of a valid CN[7]. Two valid CNs CN_1 and CN_2 are a binary decomposition of CN if and only if

- both CN_1 and CN_2 is contained in CN, and
- the set of nodes in CN_1 and CN_2 do not overlap, and
- the union of non-free tuple sets in CN_1 and CN_2 equals to the set of non-free tuple sets in CN.

For example, $CN_7 = C^Q \rightarrow U \leftarrow C^Q \rightarrow P^Q$ can be decomposed to two valid CNs: C^Q and $C^Q \rightarrow P^Q$. Note that since the node U is not a non-free tuple set, it is not included in any of the two sub-CNs.

A CN can have more than one binary decomposition. For example, there are two binary decompositions of $CN_7 = C^Q \rightarrow U \leftarrow C^Q \rightarrow P^Q$ as $\{C^Q, C^Q \rightarrow P^Q\}$ and $\{P^Q, C^Q \rightarrow U \leftarrow C^Q\}$. We can capture all such binary decompositions in a partitioning graph. A *Partitioning Graph* is a directed acyclic graph containing two kinds of nodes: *CN nodes* and *partitioning nodes* (P nodes for

[7]i.e., CNs that satisfy Rules 2 and 3 in Section II-B.

short). A CN node corresponds to a *valid* CN with respect to the given query. A partitioning node denotes a binary partitioning from a valid CN (with incoming edge) to two smaller valid CNs (with outgoing edges). A large CN node usually can be partitioned in several different ways and hence connected to several partitioning nodes. For a given P-node P, we use $P.child$ to denote its child CN (with incoming edge), and $P.parents$ to denote its parents CNs (with outgoing edges). Note that both the two parents of P are sub-CNs of $P.child$.

Example 5.2: The partitioning graph for CNs in Table III is shown in Figure 3. As an example, CN_7 has two decompositions (namely P_6 and P_7).
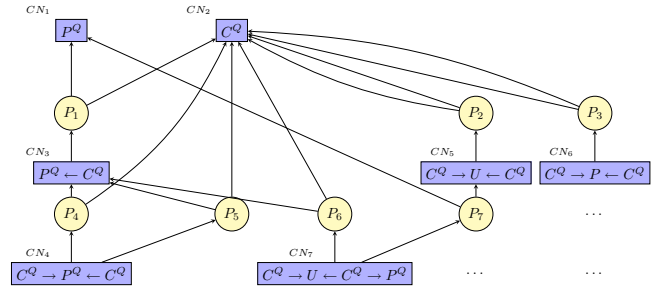
Fig. 3. Partitioning Graph

B. Query Processing

We now show the join execution of the Tree Pipeline Algorithm in Algorithm 3.

In the Block Pipeline algorithm, a block is in a state with either its USCORE or its BSCORE. In TPA, a block must be in one of the *five* states: 1) NOT_IN_QUEUE, 2) NOT_EXECUTED_USCORE, 3) NOT_EXECUTED_BSCORE, 4) EXECUTED_NOT_EMPTY, and 5) EXECUTED_EMPTY. NOT_EXECUTED_USCORE (NOT_EXECUTED_BSCORE) corresponds to USCORE (BSCORE) in the Block Pipeline Algorithm. The EXECUTED (EXECUTED_EMPTY) indicates that a block has been executed and has non-empty results (has empty results). During the execution, the status of a block moves only from a lower level to a higher level as shown in Figure 4.
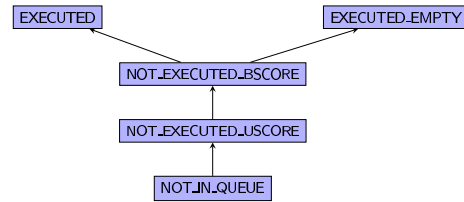
Fig. 4. Relationships Between States

The initialization procedure of the TPA algorithm is shown in Algorithm 4. All blocks in all CNs are marked as NOT_IN_QUEUE, except blocks that are the first block of a CN. In addition, when the CN is a single non-free node, we mark the block as executed, since the result is simply the corresponding tuples of the block in the non-free tuple set. (Lines 9–11). Otherwise, since there must be at least

---

Algorithm 3: Tree Pipeline Algorithm

Input : $\mathcal{CN}$ is the set of CNs
1 Initialize;   /* see Alg. 4 */;
2 **while** $top\text{-}k[k].score < Q.\text{head}().\text{getScore}()$ **do**
3    $Q.\text{empty } head \leftarrow Q.\text{pop\_max}()$;
4   **if** $head.status = $ NOT_EXECUTED_USCORE **then**
5       $head.status \leftarrow $ NOT_EXECUTED_BSCORE;
6       $Q.\text{push}(head, \text{calc\_bscore}(head))$;
7   AddNeighbors($head$);
8   **else if** $head.status = $ NOT_EXECUTED_BSCORE **then**
9       $plan \leftarrow $ FindExecPlan($head$);
10   **forall** $curP \in plan$ (in that order) **do**
11          $curC \leftarrow head.\text{projectTo}(curP.child)$;
12   AddNeighbors($curC$);
13          $curC.resultset \leftarrow$
         executeSQL(formQuery($curP$));
14   **if** $curC.resultset \neq \emptyset$ **then**
15             $curC.status \leftarrow $ EXECUTED;
16   **forall** result $t \in curC.resultset$ **do**
17                $Q.\text{push}(t, \text{calc\_score}(t))$;
18   **else**
19             $curC.status \leftarrow $ EXECUTED_EMPTY;
            /* need to propagate this to
               other relevant blocks   */
20   **forall** rest of the P-nodes, $curP$, in $plan$
            do
21   of the blocks $curC \leftarrow$
               $head.\text{projectTo}(curP.child)$;
22   AddNeighbors($curC$);
23                $curC.status \leftarrow $ EXECUTED_EMPTY;
24   **break;**
25   **else**
26   Insert $head$ into $top\text{-}k$;
27 **return** $top\text{-}k$;

Algorithm 4: Initialize

Input : $\mathcal{CN}$ is the set of CNs
1 $Q \leftarrow \emptyset$;
2 all blocks in all CNs are marked as NOT_IN_QUEUE;
3 **forall** $cn \in \mathcal{CN}$ **do**
4    $b \leftarrow$ the first block of cn;
5   **if** $\text{size}(cn) > 1$ **then**
6       $b.status \leftarrow $ NOT_EXECUTED_USCORE;
7       $Q.\text{push}(b, \text{calc\_uscore}(b))$;
8   **else**
9       $b.status \leftarrow $ EXECUTED;
10   **forall** result $t \in b$ **do**
11          $Q.\text{push}(t, \text{calc\_score}(t))$;

---

a join in the CN, we will only compute the *uscore* of the block and push it into the priority queue (Lines 6–7).

The overall flow of TPA algorithm (Algorithm 3) is similar to that of the Block Pipeline Algorithm. At each step of TPA, the first item ($head$) of the priority queue is popped out (Line 3). If its state is NOT_EXECUTED_USCORE (Lines 5–7), we calculate its *bscore*, upgrade its state to NOT_EXECUTED_BSCORE, and push its neighbors into the queue with their *uscore*. The last step is implemented in Algorithm 5. Note that the **if** test is necessary (e.g., some

block might have already been added to the queue and have already been executed).

The main difference between the TPA algorithm and the Block Pipeline Algorithm is when the current head block is in the state of NOT_EXECUTED_BSCORE (Lines 9–24 in Algorithm 3). In TPA, rather than directly executing the SQL query corresponding to the block, we find an appropriate execution plan to exploit the possibility of sharing computation or pruning. An execution plan is essentially a sequence of binary decompositions arranged from fine to coarse granularities, i.e., the last decomposition is for the current CN. The plan is found by Algorithm 6 (to be covered shortly in Section V-C) and is implemented as a sequence of partitioning nodes. For each partitioning nodes, we form an SQL query that joins the intermediate results of its two parent nodes to obtain the result for its child node (Lines 11–13). If the result is non-empty, it is pushed into the queue with their actual scores (Lines 15–17). Otherwise, we know the current block is empty, and more importantly, we need to *propagate* such information to other relevant blocks. Hence we iterate through the rest of the partitioning nodes and mark the projections of the current EXECUTED_EMPTY block as EXECUTED_EMPTY (Lines 19–23). An subtle yet important issue is that we need to invoke AddNeighbors on projected blocks (Line 22) and on the current block (Line 12), as otherwise we may miss some blocks and hence potentially some query results.

Example 5.3: Consider running the TPA algorithm to a state when the current block b is from CN_7 and its state is already NOT_EXECUTED_BSCORE. Assume the FindExecPlan procedure returns $\langle P_1, P_6 \rangle$, which essentially means the best plan to acquire the result of the current block in CN_7 is to first execute the corresponding SQL query to obtain b's projection on CN_3 (due to P_1) and then join the intermediate results (if not empty) with b's projection on CN_2 (due to P_6). If none of the intermediate results (i.e., b's projection on any of the CNs) is empty, we not only obtain the result for the current block b in CN_7, but also the result of b's projection in CN_3 (named b'). Note that the latter will also reduce the cost of obtaining other blocks, e.g., the block b'' in CN_4 such that b'''s projection on CN_3 is b'. On the other hand, if b' is empty, b's state will be directly marked as EXECUTED_EMPTY and a costly four-way join is avoided (as required by all other algorithms). Note that we won't propagate the fact that b' is empty to CNs not involved in the current execution plan (e.g., CN_4). However, when the corresponding block b'' in CN_4 becomes the head of the queue, the FindExecPlan procedure (See Section V-C) will notice that b' is in the state of EXECUTED_EMPTY and will prune b'' directly without sending any query to the database.

C. Find An Execution Plan

Given a block B of the current CN CN_c, we can utilize the partition graph to either prune the block or compute the result of B from some intermediate results (i.e., query results cached for some blocks belonging to other CNs).

Algorithm 5: AddNeighbors(B)

---

1 forall neighboring block B_n of block B **do**
2   **if** $B_n.state =$ NOT_IN_QUEUE **then**
3 Q.push(B_n, calc_uscore(B_n));
4 $B_n.state \leftarrow$ NOT_EXECUTED_USCORE;

Since a CN node may have multiple parent P-nodes, it is desirable to find the *execution plan* that has the lowest cost.

Algorithm 6: FindExecPlan(B)

Input   : A block B; the Partitioning Graph $\mathcal{P}$
Output : The best execution sequence to compute the current block B. Return $\emptyset$ if the current block is pruned.
1 $BestP \leftarrow \emptyset$;
2 if $B.status \in \{$EXECUTED, EXECUTED_EMPTY$\}$ **then**
3   **return** $\emptyset$;
4 forall $P \in B.parents()$ **do**
5   **forall** $CN \in P.parents()$ **do**
6 $B' \leftarrow B.\text{projectTo}(CN)$;
7 $B'.plan \leftarrow$ FindExecPlan(B');
8   **if** $B'.status \neq$ EXECUTED_EMPTY **then**
9 $P.plan \leftarrow P.plan \cup B'.plan$;
10   **else**
11 $B.status \leftarrow$ EXECUTED_EMPTY; /* Prune B if any of its parent node has empty resultset */;
12   **return** $\emptyset$;
             /* recursively pruning till the outmost CN   */
13 $P.plan \leftarrow P.plan \cup P$;
14   **if** $cost(P.plan) < cost(BestPlan)$ **then**
15 $BestPlan \leftarrow P.plan$;
16 return $BestPlan$;

The FindExecPlan algorithm (Algorithm 6) is designed to perform this task. It will return an execution plan in the form of an ordered list of P-nodes. We have the invariant that the corresponding blocks in the two parents of the P-nodes in the execution plan must have been executed and has non-empty results. Therefore, the plan is always viable in the sense that the result of the current block can be obtained by forming and executing SQL queries for each P-nodes in the execution plan. The algorithm may return an empty set, which indicates that the current block B can be safely prune as one of its projection is found to have empty result.

The algorithm searches the best execution plan by a breadth-first search of all the possible partitioning of the current block B. This is achieved by a loop through all the P-nodes (Line 5) and recursively call the FindExec-Plan. The plan with the lowest cost is maintained in the $BestPlan$ variable (Line 15) and will be returned at last.

Line 12 implements a *pull-based* pruning strategy. When a projected block is executed yet has empty result, instead of returning directly, we still maintain the iteration but prune the rest of the blocks by setting their status to EXECUTED_EMPTY.

Example 5.4: Consider a simplified example on the partitioning graph in Figure 3. Assume the current $head$ is a block B in CN_7 and we first consider the partitioning node P_6. Since CN_2 is a size-1 CN, all its blocks are in the EXECUTED state. If the corresponding block on CN_3 has been executed, the returned execution plan will be P_6; otherwise, FindExecPlan will recursively find execution plan for the corresponding block on CN_2. The recursive call will return plan $\langle P_1 \rangle$ (as both of its parent nodes are size-1 CN node and all blocks are in EXECUTED state), and the final execution plan for the block B on CN_7 will be $\langle P_1, P_6 \rangle$. The rest of the partitioning nodes of CN_7, P_7 in our example, will be considered in a similar fashion; it will return $\langle P_2, P_7 \rangle$. Finally, the execution plan with the least cost will be selected and returned.

D. Discussion

Although the idea of sharing common sub-expressions is a well-known topic [21] in query optimization, DIS-COVER [6] is the only work to apply this idea to the keyword search problem in relational databases. Our sharing method has the following major differences from theirs:

- *Basic Sharing Units*: the basic sharing unit in the DIS-COVER system is a join expression involving at least two relations, while the sharing unit in our method is a block in a candidate network, which is typically much smaller than a join expression in most practical top-k settings. This design choice also guarantees that the TPA algorithm only caches potential keyword query results (which might not be among the top-k results) while the DISCOVER method may cache many results not belonging to the keyword query.
- *Integration into top-k algorithms*: The sharing method in DISCOVER can be easily integrated into the Sparse algorithm [7]. However, it is hard to be integrated into algorithms such as Global Pipeline, Skyline Sweeping, or Block Pipeline, as they operate on a tuple basis. Our TPA algorithm results from integrating our sharing idea with the Block Pipeline algorithm to further improve its performance.

VI. EXPERIMENTS

In order to evaluate the effectiveness and the efficiency of proposed methods, we have conducted extensive experiments on large-scale real datasets under a number of configurations.

The datasets we used include: the Internet Movie Database (**IMDB**) [8], DBLP data (**DBLP**) [22], and Mondial [9]. All are real datasets. Schema and statistics of the two datasets can be found in Tables VI(a) and VI(b). For the IMDB dataset, we converted a subset of its raw text files into relational tables. Mondial is a very small dataset consisting only around 10K tuples about geographic

---

[8]http://www.imdb.com/interfaces
[9]http://www.dbis.informatik.uni-goettingen.de/Mondial/

TABLE V
TOP-1 RESULT FOR DQ1 (NIKOS CLIQUE) ON DBLP

Method	Size	Top-1 Result
[7]	1	**InProceeding**: Clique-to-Clique Distance Computation Using a Specific Architecture
[10]	6	**Person**: Nikos Karatzas ← **Proceeding** → **Series** ← **Proceeding** — **InProceeding**: Maximum Clique Transversals / **InProceeding**: On … Clique-Width and …
Ours	3	**Person**: Nikos Mamoulis ← **RPI** → **InProceeding**: Constraint-Based Algorithms for Computing Clique Intersection Joins

information. But its schema is much more complex than the other two datasets, which consists 28 relations. We will use Mondial only for evaluating CN Generation efficiency.

We manually picked a large number of queries for each dataset. We tried to include a wide variety of keywords and their combinations in the query sets, e.g., selectivity of keywords, size of the most relevant answers, number of potential relevant answers, etc. The complete list of queries used in the experiment can be found in [23]. We focus on a subset of the queries here. There are 22 queries for the IMDB dataset (IQ1 to IQ22) with query length ranging from 2 to 3. There are 18 queries for the DBLP dataset (DQ1 to DQ18) with query length ranging from 2 to 4.

TABLE VI
DATASET STATISTICS (TEXT ATTRIBUTES ARE UNDERLINED)

(a) IMDB Dataset

Relation Schema	# Tuples
movies(mID, name)	833,512
direct(mID, dID)	561,173
directors(dID, name)	121,928
actressplay(asID, charactor, mID)	2,262,149
actresses(asID, name)	445,020
actorplay(atID, charactor, mID)	4,244,600
actors(atID, name)	741,449
genres(mID, genre)	629,195
Total Number of Tuples	**9,839,026**

(b) DBLP Dataset

Relation Schema	# Tuples
InProceeding(InProceedingId, Title, Pages, URL, ProceedingId)	212,273
Person(PersonId, Name)	174,709
RelationPersonInProceeding(InProceedingId, PersonId)	491,777
Proceeding(ProceedingId, Title, EditorId, PublisherId, SeriesId, Year, Url)	3,007
Publisher(PublisherId, Name)	86
Series(SeriesId, Title, Url)	24
Total Number of Tuples	**881,867**

We used two popular relational database servers, both with their default configurations. Indexes were built on all primary key and foreign key attributes. For most of the queries, similar results were obtained on the two systems.

We implemented the **Sparse** and global pipeline (**GP**) algorithms, and our skyline sweep (**SS**), block pipeline algorithms (**BP**) and tree pipeline (**TP**). Note that we can lower bound the execution time of the **Hybrid** algorithm [7] as the minimum of the running times of Sparse and GP.

Without this optimization, the original GP algorithm would have sent an excessive number of queries to the database and incurred significant overhead. Unless specified explicitly, all algorithms ran using OR semantics.

All algorithms were implemented using JDK 1.5 and JDBC. All experiments were run on a PC with a 1.8GHz CPU and 512M memory running Debian GNU/Linux 3.1. The database server and the client were run on the same PC. All algorithms were run in warm buffer mode and Java JIT was enabled.

To measure the effectiveness, we adopt two metrics used in the previous study [10]: (a) number of top-1 answers that are relevant (**#Rel**), and (b) reciprocal rank (**R-Rank**). In order to select the relevant answer, we ran all the algorithms for the same query and merged their top-20 results. Then we manually judged and picked the relevant answer(s) for each query. The relevant answer(s) must satisfy two conditions: it must match all the search keyword and its size must be the smallest. For example, the manually marked relevant answer for the query "nikos clique" is a paper named "Constraint-Based Algorithms for Computing Clique Intersection Joins" written by "Nikos Mamoulis". When measuring the reciprocal rank, we search for the first relevant answer in the top-20 results. In case none of the top-20 answers is relevant, we upper bound its R-Rank value by $\frac{1}{\#uniq\_score+1}$, where $\#uniq\_score$ is the number of unique scores in its top-20 results. To measure efficiency, we measure the average elapsed times of the algorithms over several runs.

A. Effectiveness

We show the *reciprocal rank*s of [7], [10], and our proposed method on the DBLP dataset in Table VII. For [10], we used all four normalizations, but not the phrase-based ranking. For our ranking method, we vary the tuning parameter p from 1.0 to 2.0, thus representing the change of preference from the OR-semantics to the AND-semantics. The results show that our R-Rank is higher than other methods, as our method returns the relevant result as the top-1 result for 16 out of the 18 DBLP queries when $p = 1.0$. While [7] and [10] methods have a tie in #Rel measure, [10] actually performs better than [7], because it often returns relevant answer(s) within the top-5 results, while [7] method often fails to find any relevant answer in the top-20 results. This is reflected in their R-Rank measures. Similar results were obtained on the IMDB dataset too.

Manual inspection of the top-20 answers returned by the algorithms reveals some interesting "features" of the

ranking methods. Due to the inherent bias in [7]'s ranking aggregation method and extremely harsh penalty on the CN sizes, it tends to return results that have only partial matches to the query or small-sized results. [10] proposed using a soft CN size normalization and a non-linear rank aggregation method. Consequently, it tends to return large-sized results that match most of the keywords. Our method seems to strike a good balance between the completeness of the matches and size of the results. For instance, we show the top-1 results returned by all ranking methods for DQ1 on DBLP in Table V.

TABLE VII
EFFECTIVENESS ON THE DBLP DATASET BASED ON TOP-20 RESULTS

	[7]	[10]	$p = 1.0$	$p = 1.4$	$p = 2.0$
#Rel	2	2	16	16	18
R-Rank	≤ 0.243	≤ 0.333	0.926	0.935	1

TABLE VIII
p'S IMPACT ON R-RANK

Dataset	QueryID	$p = 1$	$p = 1.4$	$p = 2.0$
DBLP	DQ9	1/3	1/2	1
DBLP	DQ17	1/3	1/3	1
IMDB	IQ10	1	1	1
IMDB	IQ17	1/3	1/3	1
IMDB	IQ19	1/2	1	1
IMDB	IQ21	1/2	1/2	1/2

We also conducted experiments by varying p from 1.0 to 2.0. This should inject more AND semantics into our ranking method. As the default $p = 1.0$ already returns relevant results for most queries, we only list queries whose result qualities (R-Rank values) are affected by the varying p in Table VIII. With an increasing value of p, the R-Rank values for most such queries increase. This is because we start to penalize more on results that does not match all the keywords. For example, when $p = 1.0$, the relevant answer for DQ9 is only ranked as the third. The top-1 answer matches all but one keyword. When p increases to 1.4, the relevant answer moves up to the second. Finally, when p reaches 2.0, it is successfully ranked as the top answer.
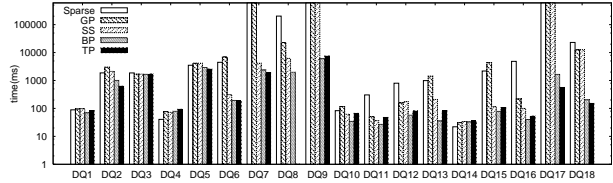
B. Efficiency

We show running time for all queries on the DBLP and IMDB datasets in Figures 5(a) to 5(e) for $k = 20$, $k = 10$ and $k = 1$. Note that the y-axis is in logarithm scale. We can make the following observations:
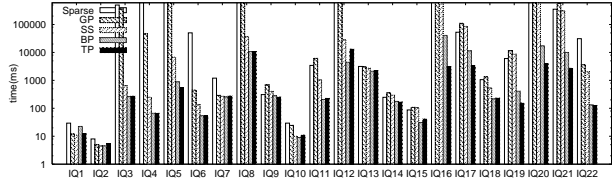
- BP and TP are usually the faster algorithm on both DBLP and IMDB datasets. The speedup is most substantial on *hard queries*, e.g., DQ7, DQ13, and DQ17. BP can achieve up to two orders of magnitude speedup against the better algorithm of Sparse and GP (thus the lower bound of Hybrid algorithm). BP can return top-10 answers within 2 seconds for 89% of the queries on the DBLP dataset and 77% queries on the IMDB dataset which is 10 times larger.
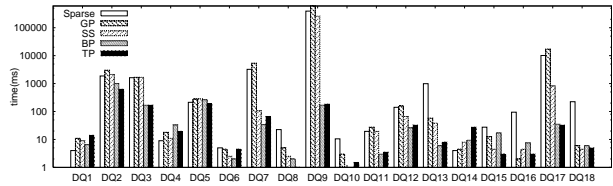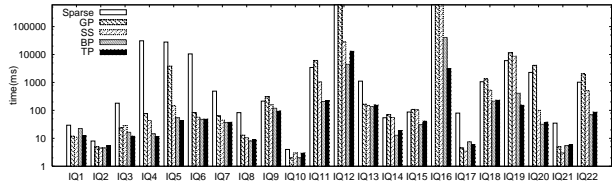

(a) Query Time, DBLP, $k = 20$, the i-th Query is DQi.

(b) Query Time, DBLP, $k = 10$, the i-th Query is DQi.

(c) Query Time, IMDB, $k = 10$, the i-th Query is IQi.

(d) Query Time, DBLP, $k = 1$, the i-th Query is DQi.

(e) Query Time, IMDB, $k = 1$, the i-th Query is IQi.

Fig. 5. Evaluation of Query Processing Performance – I

- On hard queries, TP usually outperforms BP. While on *easy* queries or small k, performance of BP and TP are similar. The reason is that top-k results can be retrieved from small number of CNs, and TP can not take advantage by sharing computation among CNs. Also note that on DQ8 query, TP fails to return top-10 results, due to the limitation of the number of temporary relations allowed to be created in MySQL.
- SS usually outperforms Sparse and GP, with only a few losses to Sparse, even for $k = 20$. When k is small, SS shows more performance advantages. SS can achieve up to one order of magnitude speedup against the better algorithm of Sparse and GP.
- There is no sure winner between Sparse and GP. In general, while Sparse might lose for small k values or *easy* queries, its performance does not deteriorate too
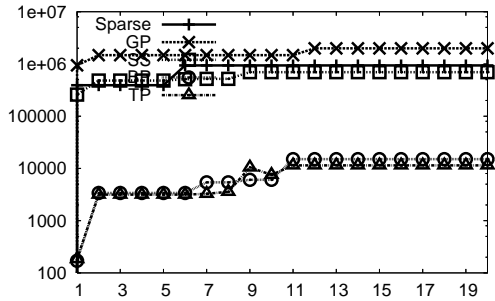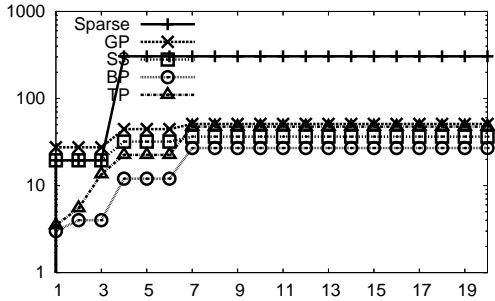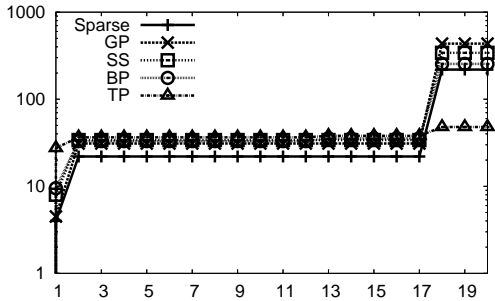
(a) DBLP, DQ9, $k = 1$ to 20

(b) DBLP, DQ11, $k = 1$ to 20

(c) DBLP, DQ14, $k = 1$ to 20

Fig. 6.   Evaluation of Query Processing Performance – II

(a) Query Time, DBLP, $k = 1$, AND Semantics

(b) Query Size Ratio, DBLP, $k = 20$, the i-th Query is DQi.

(c) Query Number, DBLP, $k = 20$, the i-th Query is DQi.

(d) Ratio of DBMS Query Processing Time over Overall Elapsed Time, DBLP, $k = 20$, the i-th Query is DQi.

Fig. 7.   Evaluation of Query Processing Performance – III

much for large k or *hard* queries.

- All algorithms are more responsive for smaller k values. We note that since our ranking function usually returns the relevant answer as the top-1 answer, the execution time for top-1 answer is an important indicator of system performance from the user's perspective.

We plotted the execution times with different k values for all queries. We selected three representative figures from the DBLP query set and show the results in Figures 6(a) to 6(c). In general, the costs of all algorithms increase with the increasing k value, as more candidate answers need to be found and compared. Some of the queries are amenable to top-k optimized algorithms (e.g., DQ11), where the other four algorithms all perform significantly better than Sparse. There are also queries where BP and TP perform better than the others (e.g. DQ9). DQ14 shows an example that when k grows large, TP takes advantage by sharing intermediate results, while all other algorithms jump to a large running cost.
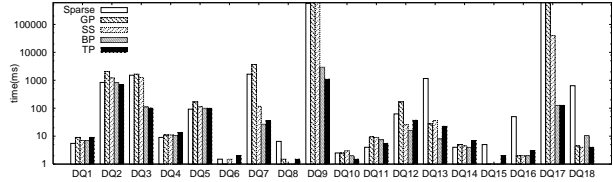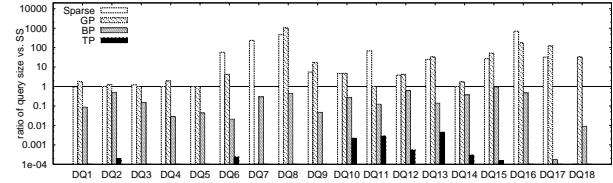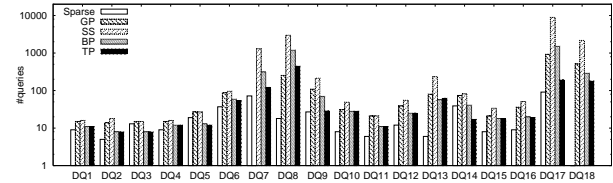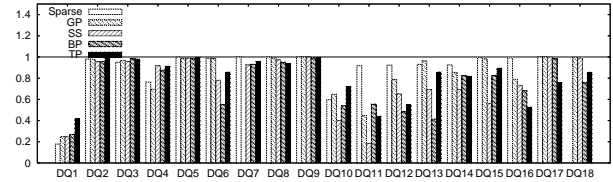
We also run all the algorithm using the AND semantics for top-1 results on DBLP. All algorithms can find the relevant results as the top-1 results, so we focus on the execution time, which is plot in Figure 7(a). It is obvious that similar conclusions can be drawn about the relative performance of the algorithms.

The fundamental reason of superior performances of our algorithms is that they avoid many unnecessary database probes. To verify this, we recorded the number of candidates each algorithm has checked against the database (QSize) and the number of queries sent to the database (QNum) and plot them in Figures 7(b) and 7(c), respectively. Specifically, we choose SS as the baseline algorithm (since it has the minimal QSize without utilizing a second upper bounding function) and calculate the ratio of probes of other algorithms over this baseline number. We can observe from Figure 7(b) that Sparse usually has to examine many candidates, as it cannot stop earlier until the complete query of a CN has been executed. GP is only slightly better than Sparse, partly because we specify a rather large k value, hence GP's performance drops quickly. BP algorithm makes use of additional upper bounding functions and delays probing the database as much as possible. As

a result, it usually examines fewer candidates than SS. The number of candidates examined by TP is extremely small, and for many queries, the TP bar can not been seen in the plotted graph. But somehow the measurement is unreasonable, as TP partitions the search space into lower-dimensions, and the number of candidates is counted in the low dimensional space. In terms of query numbers, as expected, Sparse always sends a small number of queries. Interestingly, SS sends the largest number of queries in all the cases compared to GP, BP and TP. The reason is that the three algorithms can examine a number of candidates together in one single query using our range parametric query optimization; in contrast, SS exams candidates in an ad-hoc manner and such optimization cannot be applied.

We broke down the elapsed time for all the four algorithms. We summed up time used by the RDBMS to process queries and divided them over the total elapsed times of the queries. The result for the DBLP dataset is shown in Figure 7(d). Overall speaking, the dominant cost for all algorithms is the DBMS query processing time. GP's cost is mostly dominated by DBMS query processing time, as GP only needs to keep a few data structures (the current tuple in each of the non-free tuple set of the CNs) and does not have expensive calculation and data structure maintenance overhead. Among the other algorithms there is no clear winner. Sparse's overhead mainly comes from the need to calculate the IR scores for *all* results returned by its large-sized queries. SS needs to spend time on maintaining the priority queue. TP needs to maintain the partitioning graph and to find the best plan. Overall speaking, the BP algorithm is still dominated by DBMS query processing time (averaged about 71%), but to the least extent. This is because, intuitively, BP spends more time in its internal calculation (of upper bounding scores) to avoid expensive database probes.

VII. RELATED WORKS

Keyword Search Systems   The main goal of a keyword search system is to find a set of closely inter-connected tuples that collectively match the keywords.

One type of approaches is based on modeling data as a graph, and the results as subtrees or sub-graphs. The keyword search problem can be shown to be an instance of the Group Steiner tree problem, which is NP-hard. Exhaustive search based on dynamic programming is developed [24], [25], which is capable to find the optimal solution of minimum group Steiner tree problem with polynomial time when the number of keywords(groups) is fixed. Most other work relax the definition or adapt a heuristic approach to attain reasonable efficiency. Early work in this category includes RUI [26] and the BANKS systems [3], [8]. Q-subtree [27] combines the technique of BANKS and [28]. More recently, re-computing [29] and indexing approaches [30]–[34] are also developed for the efficiency propose.

Another type of approaches is based on relational databases where structured data are stored. As such, they exploit the schema information and leverage the DBMS for query processing. Early work includes DBXplorer [2]. When data schema is used for query processing purpose, existing approach is mainly based on Candidate Network (CN) generation, Different query processing techniques are then applied immediately [2], [6], [7], [35], [36] or in an interactive way [29], [37]. Data schema is also used for indexing [16], ranking [10], and data browsing and user interface design [24], [38].

 [39] provides a comprehensive survey on recent survey covering keyword query on both structured and semi-structured databases.

Besides the common definition of keyword search, studies have also been performed on identifying entities that do not match keywords directly, but implicitly relevant or "near" the occurrence of search keywords [5], [40], [41]. Keyword search can also be an idea means to associate structured data with unstructured data [42], [43].

Ranking and Searching Quality   Keyword searches are inherently ambiguous, and not all query results are equally relevant to a user. Most work focuses on bringing more effective ranking from IR literatures. Various ranking schemes have been proposed to order the query results into a sorted list so users can focus on the top results. Various ranking schemes are used in existing work, which consider both the properties of data nodes/tuples such as TF*IDF, node weight, or page-rank style ranking, and inter-tuple or global properties such as number of edges, weights on edges, size normalization, or redundancy penalty. Some advanced features, such as schema term awareness and phrase-based ranking, are also proposed [10], [24], [25], [27], [28], [30], [32], [33], [35], [38], [44]–[49].

Keyword search has been studied under a few generalized contexts too [45], [46], [50]. In order to help users to find interesting results and to improve search quality, techniques of result browsing and clustering [24], [51], [52], query cleaning and suggestion [53]–[55] are also developed.

Rank Aggregation   Given a set of objects, each being scored according to some aggregate function on its attributes, the rank aggregation query considers the problem of retrieving the k objects with the highest scores.

Rank aggregation query processing has also been extensively studied in the literature. Fagin *et al.* [11], [56] introduced a set of novel algorithms, assuming sorted access and/or random access of the objects is available on each attribute. A number of improvements have been suggested after Fagin's seminal work, for example, minimizing computational cost [57], and minimizing the IO cost [58]. Some other approaches to attack the problem include building indexes [59] or building materialized views [60].

A generalized version of rank aggregation problem is to consider more general or expensive predicates as the underlying ranked objects. Expensive predicates checking for top-k query has been studied [13], [61]–[63] to support user-defined functions, external predicates, fuzzy joins, etc. As the bottleneck of proceeding these queries comes from score predicates rather than finding object candidates, these papers focus on reducing the number of predicates to be

made.

[12] studied finding top-k joined objects and proposed a J^* algorithm, which is based on the A^* class of search algorithms. A similar probing mechanism can be found in the work from Chang *et al.* [13], where an optimal algorithm, MPro, was proposed based on the *necessary probing principle*. A number of approaches were suggested by Iiyas *et al.* [64], including *nested-loop rank-join* (NRJN) and *hash rank-join* (HRJN), which can be viewed as variant of the ripple join algorithm [14]. More recently, [65] gives a more general problem statement for the rank join query, analyzes existing techniques, and studies the theoretical aspects of the problem.

VIII. Conclusions

In this paper, we studied supporting effective and efficient top-k keyword queries over relational databases. We proposed a new ranking method that adapts the state-of-the-art IR ranking function and principles into ranking trees of joined database tuples. Our ranking method also has several salient features over existing ones. We also studied query processing method tailored for our non-monotonic ranking functions. Three algorithms were proposed that aggressively minimize database probes and maximize computational sharing. We have conducted extensive experiments on large-scale real databases. The experimental results confirmed that our ranking method could achieve high precision with high efficiency to scale to databases with tens of millions of tuples.
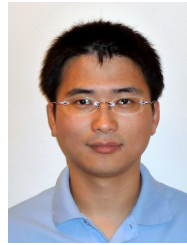
Acknowledgment

References

[1] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, "Integrating DB and IR technologies: What is the sound of one hand clapping?" in *CIDR*, 2005, pp. 1–12.

[2] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A system for keyword-based search over relational databases." in *ICDE*, 2002, pp. 5–16.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS." in *ICDE*, 2002, pp. 431–440.

[4] B. Kimelfeld and Y. Sagiv, "Efficient engines for keyword proximity search." in *WebDB*, 2005, pp. 67–72.

[5] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina, "Proximity search in databases," in *VLDB*, 1998.

[6] V. Hristidis and Y. Papakonstantinou, "DISCOVER: Keyword search in relational databases." in *VLDB*, 2002, pp. 670–681.

[7] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-Style Keyword Search over Relational Databases," in *VLDB*, 2003.

[8] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases." in *VLDB*, 2005, pp. 505–516.

[9] B. Kimelfeld and Y. Sagiv, "Finding and approximating top-k answers in keyword proximity search." in *PODS*, 2006, pp. 173–182.

[10] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases." in *SIGMOD*, 2006, pp. 563–574.

[11] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware." in *PODS*, 2001.

[12] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs." in *VLDB*, 2001, pp. 281–290.

[13] K. C.-C. Chang and S. Hwang, "Minimal probing: supporting expensive predicates for top-k queries." in *SIGMOD*, 2002, pp. 346–357.

[14] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *SIGMOD 1999*, 1999, pp. 287–298.

[15] S. E. Robertson, H. Zaragoza, and M. J. Taylor, "Simple BM25 extension to multiple weighted fields." in *CIKM*, 2004, pp. 42–49.

[16] Q. Su and J. Widom, "Indexing relational database content offline for efficient keyword-based search," in *IDEAS*, 2005.

[17] R. Wilkinson, J. Zobel, and R. Sacks-Davis, "Similarity measures for short queries." in *TREC*, 1995.

[18] D. E. Rose and D. R. Cutting, "Ranking for usability: Enhanced retrieval for short queries," Apple Technical Report, Tech. Rep. 163, 1996.

[19] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Communication of the ACM*, vol. 26, no. 11, pp. 1022–1036, 1983.

[20] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator." in *ICDE*, 2001, pp. 421–430.

[21] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[22] R. Cyganiak, "D2RQ benchemarking," http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2rq/benchmarks/.

[23] Y. Luo, X. Lin, W. Wang, and X. Zhou, "SPARK: Top-k keyword query in relational databases," School of Computer Science and Engineering, University of New South Wales, Tech. Rep. 0708, 2007.

[24] S. Wang, Z. Peng, J. Zhang, L. Qin, S. Wang, J. X. Yu, and B. Ding, "Nuits: A novel user interface for efficient keyword search over databases," in *VLDB*, 2006, pp. 1143–1146.

[25] 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*, 2007.

[26] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal, "Query relaxation by structure and semantics for retrieval of logical web documents," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 4, pp. 768–791, 2002.

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

[28] B. Kimelfeld and Y. Sagiv, "Finding and approximating top-k answers in keyword proximity search," in *PODS*, 2006, pp. 173–182.

[29] A. Baid, I. Rae, A. Doan, and J. Naughton, "Toward Industrial-Strength Keyword Search Systems over Relational Data," in *ICDE*, 2010.

[30] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword search on external memory data graphs," *PVLDB*, vol. 1, no. 1, pp. 1189–1204, 2008.

[31] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina, "Proximity search in databases," in *VLDB*, 1998, pp. 26–37.

[32] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *SIGMOD*, 2007, pp. 305–316.

[33] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *SIGMOD*, 2008.

[34] A. Markowetz, Y. Yang, and D. Papadias, "Reachability Indexes for Relational Keyword Search," in *ICDE*, 2009.

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

[36] L. Qin, J. X. Yu, and L. Chang, "Keyword Search in Databases: The Power of RDBMS," in *SIGMOD*, 2009.

[37] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton, "Combining Keyword Search and Forms for Ad Hoc Querying of Databases," in *SIGMOD*, 2009.

[38] P. Wu, Y. Sismanis, and B. Reinwald, "Towards keyword-driven analytical processing," in *SIGMOD*, 2007, pp. 617–628.

[39] J. X. Yu, L. Qin, and L. Chang, *Keyword Search in Databases*. Morgan & Claypool, 2009.

[40] T. Grabs, K. Böhm, and H.-J. Schek, "PowerDB-IR: Information retrieval on top of a database cluster." in *CIKM*, 2001, pp. 411–418.

[41] Y. Tao and J. X. Yu, "Finding Frequent Co-occurring Terms in Relational Keyword Search," in *EDBT*, 2009.

[42] P. Roy, M. K. Mohania, B. Bamba, and S. Raman, "Towards automatic association of relevant unstructured content with structured query results," in *CIKM*, 2005.

[43] M. Bhide, A. G. 0004, R. Gupta, P. Roy, M. K. Mohania, and Z. Ichhaporia, "Liptus: associating structured and unstructured information in a banking environment," in *SIGMOD Conference*, 2007.

[44] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD*, 2003.

[45] G. Li, J. Feng, J. Wang, and L. Zhou, "An effective and versatile keyword search engine on heterogenous data sources," *PVLDB*, vol. 1, no. 2, pp. 1452–1455, 2008.

[46] M. Sayyadan, H. LeKhac, A. Doan, and L. Gravano, "Efficient keyword search across heterogeneous relational databases," in *ICDE*, 2007.

[47] S. Tata and G. M. Lohman, "SQAK: doing more with keywords," in *SIGMOD*, 2008, pp. 889–902.

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

[49] B. Yu, G. Li, K. R. Sollins, and A. K. H. Tung, "Effective keyword-based selection of relational databases," in *SIGMOD*, 2007, pp. 139–150.

[50] G. Koutrika, A. Simitsis, and Y. Ioannidis, "Précis: The essence of a query answer," in *ICDE*, 2006.

[51] G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina, "DataClouds: Summarizing Keyword Search Results over Structured Data," in *EDBT*, 2009.

[52] Z. Liu, P. Sun, and Y. Chen, "Structured Search Result Differentiation," in *VLDB*, 2009.

[53] K. Q. Pu and X. Yu, "Keyword query cleaning," *PVLDB*, vol. 1, no. 1, pp. 909–920, 2008.

[54] N. Sarkas, N. Bansal, G. Das, , and N. Koudas, "Measure-driven Keyword-Query Expansion," in *VLDB*, 2009.

[55] S. Chaudhuri and R. Kaushik, "Extending Autocompletion to Tolerate Errors," in *SIGMOD*, 2009.

[56] R. Fagin, "Combining fuzzy information from multiple systems." *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.

[57] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung, "Efficient aggregation of ranked inputs." in *ICDE*, 2006.

[58] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "IO-Top-k: Index-access optimized top-k query processing." in *VLDB*, 2006, pp. 475–486.

[59] D. Xin, C. Chen, and J. Han, "Towards robust indexing for ranked queries." in *VLDB*, 2006, pp. 235–246.

[60] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views." in *VLDB*, 2006, pp. 451–462.

[61] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates." in *SIGMOD*, 1993, pp. 267–276.

[62] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimizing disjunctive queries with expensive predicates." in *SIGMOD*, 1994, pp. 336–347.

[63] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases." in *ICDE*, 2002, pp. 369–.

[64] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases." *VLDB Journal*, vol. 13, no. 3, pp. 207–221, 2004.

[65] K. Schnaitter and N. Polyzotis, "Evaluating rank joins with optimal cost," in *PODS*, 2008, pp. 43–52.

Wei Wang is currently a Senior Lecturer at the School of Computer Science and Engineering at University of New South Wales, Australia. He received his Ph.D. degree in Computer Science from Hong Kong University of Science and Technology in 2004. His research interests include integration of database and information retrieval techniques, similarity search, and query processing and optimization.

Xuemin Lin is a Professor in the School of Computer Science and Engineering, the University of New South Wales. Dr. Lin got his PhD in Computer Science from the University of Queensland in 1992 and his BSc in Applied Math from Fudan University in 1984. He currently is an associate editor of ACM Transactions on Database Systems. His current research interests lie in data streams, approximate query processing, spatial data analysis, and graph visualization.

Xiaofang Zhou is a Professor of Computer Science at the University of Queensland. His research focus is to find effective and efficient solutions for managing, integrating and analyzing very large amount of complex data for business, scientific and personal applications. He has been working in the area of spatial and multimedia databases, data quality, high performance query processing, Web information systems and bioinformatics.

Jianmin Wang graduated from Peking University, China, in 1990, and got his M.E. and Ph.D. in Computer Software from Tsinghua University, China, in 1992 and 1995, respectively. His Ph D. thesis was awarded by the Beijing Computer Association in 1996. He is now a full professor at the School of Software, Tsinghua University. Now his research interests include: unstructured data management, workflow and BPM technology, benchmark for database system, software watermarking, and mobile digital right management.

Yi Luo received her Masters and PhD degree in Computer Science from the University of New South Wales, Australia in 2005 and 2009. She is currently in a post-doctoral position in the Laboratory Le2i of CNRS in France. Her research interests include integration of database and information retrieval techniques, semantic search, and query processing and optimization.

Keqiu Li received the Bachelor's and Master's degrees from the Department of Applied Mathematics at the Dalian University of Technology in 1994 and 1997, respectively. He received the PhD degree from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology, in 2005. He is currently a professor in the School of Computer Science and Technology, Dalian University of Technology, China. His research interests include Internet technology, networking, multimedia applications and bioinformatics.