

# SPARK: Top- $k$ Keyword Query in Relational Databases\*

Yi Luo   Xuemin Lin   Wei Wang  
University of New South Wales  
Australia  
{luoyi, lxue, weiw}@cse.unsw.edu.au

Xiaofang Zhou  
University of Queensland  
Australia  
zxf@itee.uq.edu.au

## 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*. Compared with previous approaches, our new ranking method is simple yet effective, and agrees with human perceptions. We also study efficient query processing methods for the new ranking method, and propose algorithms that have minimal accesses to the database. 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.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Query Processing; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

## General Terms

Algorithms, Experimentation, Performance

## Keywords

top- $k$ , keyword search, relational database, information retrieval

## 1. INTRODUCTION

Integration of DB and IR technologies has been an active research topic recently [6]. One fundamental driving force is the fact that more and more text data are now stored

\*This work was supported by ARC Grant DP0666428 and UNSW Goldstar Grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 11–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

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 [1, 3]. Supporting such feature has a 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 and database schema. Thirdly, it helps to reveal interesting or unexpected relationships among entities [19]. 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](http://imdb.com), and failed to find relevant answers (Figure 1). In contrast, the same search on our system (on a database populated with [imdb.com](http://imdb.com)'s data) will return results shown in Table 1, where relevant tuples from multiple relations (marked in bold font) are joined together to form a meaningful answer to the query.



Figure 1: Searching “2001 hanks” on [imdb.com](http://imdb.com)

There has been many related work dedicated to keyword search in databases recently [12, 1, 16, 15, 3, 18, 19, 20, 22]. Among them, [15] 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.* [22] by using several refined weighting schemes. BANKS [3] and BANKS2 [18] took another approach by modelling the

**Table 1: Top-3 Search Results on Our System**

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

database content as a graph and proposed sophisticated ranking and query execution algorithms. Recently, [19, 20] studied theoretical aspects of efficient query processing for top- $k$  keyword queries.

Despite the previous 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 rank aggregation function. Therefore, we also study efficient query processing methods optimized for our non-monotonic ranking 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 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 two algorithms, *skyline sweeping* and *block 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.

Similar to previous work [1, 15, 22], our system is designed to work on top of a relational DBMS that supports full-text query and inverted indexes. Our current implementation supports both Oracle and MySQL. The rationale for choosing this architecture over other alternatives (such as graph-

based method [3, 18, 9]) is that (a) Relational databases are widely used in enterprises and over the Internet, and a sizeable percentage of information stored inside RDBMSs is text; (b) it is easier to exploit many features offered by RDBMSs, e.g., data storage and recovery, index building and maintenance, and sophisticated query processing and optimization capabilities. Nonetheless, we believe our research into the RDBMS-based approach is complementary to alternative approaches.

The rest of the paper is organized as follows: Section 2 provides an overview of the problem and existing solutions. Section 3 presents our new ranking method and Section 4 introduces two query processing algorithms optimized for efficient top- $k$  retrieval. Experimental results are reported in Section 5. We introduce related work in Section 6 and Section 7 concludes the paper. A full version of the paper can be found in [23].

## 2. PRELIMINARIES

### 2.1 Problem Overview and Problem Definition

We consider a relational schema  $\mathcal{R}$  as a set of relations  $\{R_1, R_2, \dots, R_{|\mathcal{R}|}\}$ . 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.,  $Q = \{w_1, w_2, \dots, w_{|Q|}\}$ ; 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 [15].

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)* [16]. 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 1.* In this paper, we use the same running example as the previous work [15] (shown in Figure 2).

In the example,  $\mathcal{R} = \{P, C, U\}$ .<sup>1</sup> 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 “**mactor netvista**”.

<sup>1</sup>Initials of relation names are used as shorthands (except that we use  $U$  to denote *Customers*).

COMPLAINTS					
rid	prodId	custID	date	comments	
c <sub>1</sub>	p121	c3232	6-30-2002	disk crashed after just one week of moderate use on an IBM <u>Netvista</u> X41	
c <sub>2</sub>	p131	c3131	7-3-2002	lower-end IBM <u>Netvista</u> caught fire, starting apparently with disk	
c <sub>3</sub>	p131	c3143	8-3-2002	IBM <u>Netvista</u> unstable with <u>Maxtor</u> HD	

PRODUCTS			
rid	prodId	manufacturer	model
p <sub>1</sub>	p121	<u>Maxtor</u>	D540X
p <sub>2</sub>	p131	IBM	<u>Netvista</u>
p <sub>3</sub>	p141	Tripplite	Smart 700VA

CUSTOMERS			
rid	custId	name	occupation
u <sub>1</sub>	c3232	John Smith	Software Engineer
u <sub>2</sub>	c3131	Jack Lucas	Architect
u <sub>3</sub>	c3143	John Mayer	Student

Figure 2: A Running Example from [15] (Query is “maxtor netvista”; Matches are Underlined)

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  $C^Q$ ; 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.

## 2.2 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.* 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 [16] 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 2).

**Rule 1** Prune duplicate CNs.

**Rule 2** Prune non-minimal CNs, i.e., CNs containing at least one leaf 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  $R^Q$ .

Table 2: 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	$CN_1$	$P^Q$	<b>Y</b>	
1	$CN_2$	$C^Q$	<b>Y</b>	
2	$CN_3$	$P^Q \leftarrow C^Q$	<b>Y</b>	
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	$CN_4$	$C^Q \rightarrow P^Q \leftarrow C^Q$	<b>Y</b>	
3		$U \leftarrow C^Q \rightarrow U$	n	Rules (2, 3)
4	⋮	⋮	⋮	

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 [15] 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.<sup>2</sup> 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, 25, 5].

<sup>2</sup>In this paper, we name the system in [15] as DISCOVER2. A hybrid algorithm that selects either sparse or global pipeline algorithm for a query based on selectivity estimation is also proposed in [15]. It is discussed and compared with in Section 5.

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].

### 2.3 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 4.

## 3. 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 [1, 16]. DISCOVER2 later proposed a ranking formula based on the state-of-the-art IR scoring function [15]. More recently, several sophisticated improvements to the ranking formula in [15] have been suggested [22].

In this section, we first motivate our work by presenting observations that reveal several problems in the existing schemes. We then discuss our solutions to address them.

### 3.1 Problems with Existing Ranking Functions

The basic idea of the ranking method used in DISCOVER2 [15] (and its variant [22]) 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.<sup>3</sup>

For example, the IR-style ranking function used in DISCOVER2 is adapted from the TF-IDF ranking formula as:<sup>4</sup>

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

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

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

$tf_w(t)$  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.

<sup>3</sup>The aggregation function used in [22] is not monotonic. However, query processing issues with this non-monotonic aggregation function are not discussed.

<sup>4</sup>To obtain the final score of a JTT,  $score(T, Q)$  needs to be further normalized by  $T$ ’s size, i.e., multiple another  $\frac{1}{size(T)}$ .

**Table 3: Different Scoring Functions Produces Different Rankings** ( $\ln(idf_{\text{maxtor}}) = \ln(idf_{\text{netvista}}) = 1.0$  and  $dl_t = avdl_t$ )

CN	$t \in CN$	$tf_{\text{maxtor}}$	$tf_{\text{netvista}}$	$Score_t$	$Score_T$	Our Score
$c_3 \rightarrow p_2$	$c_3$	1	1	2.0	3.0	1.13
	$p_2$	0	1	1.0		
$c_1 \rightarrow p_1$	$c_1$	0	1	1.0	2.0	0.98
	$p_1$	1	0	1.0		
$c_2 \rightarrow p_2$	$c_2$	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 2. The query is “**maxtor netvista**”. Let us consider the CN:  $C^Q \rightarrow P^Q$ . If the CN is executed completely, it will produce 3 results. In Table 3, we list the detailed steps to obtain the scores ( $Score_T$ ) according to the above-mentioned method. For example,  $c_3 \rightarrow p_2$  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  $1 + \ln(1 + \ln(tf_{\text{maxtor}}(c_3))) + 1 + \ln(1 + \ln(tf_{\text{netvista}}(c_3))) = 2.0$ , and  $score(p_2, Q) = 1 + \ln(1 + \ln(tf_{\text{netvista}}(p_2))) = 1.0$ . 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  $c_1 \rightarrow p_2$  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 3.3 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 [26]. 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).

### 3.2 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 modelled 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 [30]. Our definition differs from [30] 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 complains as [30] 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

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

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

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

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,  $tf_{\maxtor}(c_2 \rightarrow p_2) = 0$ ,  $tf_{\text{netvista}}(c_2 \rightarrow p_2) = 2$ , while  $tf_{\maxtor}(c_1 \rightarrow p_1) = 1$ ,  $tf_{\text{netvista}}(c_1 \rightarrow p_1) = 1$ . 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.<sup>5</sup>

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. Therefore, we resort to an approximate solution: 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  $CN^* = R_1 \bowtie R_2 \bowtie \dots \bowtie R_l$ , and denote the percentage of tuples in  $R_j$  that matches at least a keyword  $w$  as  $p_w(R_j)$ . We can derive

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

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  $\sum_j avdl_{R_j}$ . From our preliminary experimental study [23], these approximations usually have acceptable accuracy (within 30% relative error) for small CNs of size up to 3. It is our future work to study more robust estimation techniques or alternative approaches.

### 3.3 Other Ranking Factors

**Completeness Factor.** As motivated in Section 3.1, 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

<sup>5</sup> $c_3 \leftarrow p_2$  will have score 1.13, which still makes it ranked as the first result.

intuition has been recognized by IR researchers when studying ranking for *short queries* [31, 27].

Our proposed completeness factor is derived from the *extended Boolean model* [28]. 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 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  $P_{ideal} = \underbrace{[1, \dots, 1]}_m$ . 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  $\infty$ . To see that, consider  $p \rightarrow \infty$ , 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 [1] or for the OR semantics [15].

**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  $\frac{1}{size(CN)}$  [15] usually penalizes too much for even moderate-sized CNs. We experimentally found that the following size normalization factor works well in the experiment:

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

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.

### 3.4 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)). The final score of the JTT,  $score(T, Q)$ , is the product of all the three scores:

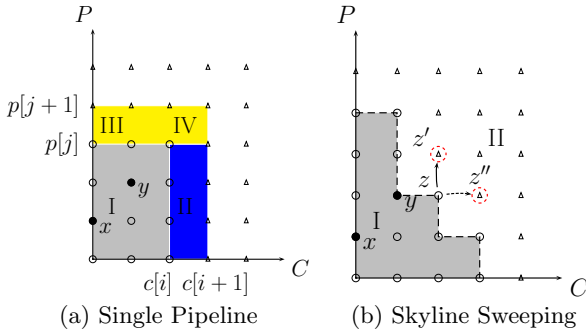
$$score(T, Q) = score_a(T, Q) \cdot score_b(T, Q) \cdot score_c(T, Q)$$

## 4. Top- $k$ JOIN ALGORITHM

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.

### 4.1 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 [15] to illustrate the challenge and motivate our algorithms.



**Figure 3: Query Evaluation Strategies**

*Example 4.* Figure 3(a) illustrates a snapshot of running the single pipeline algorithm on the CN  $C \rightarrow P$ . 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  $c[i] \rightarrow p[j]$ , 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 3. If we assume  $idf_{netvista} > idf_{maxtor}$ , then  $score(c_2) = score(c_1)$  but  $score(p_2) > score(p_1)$ . However, we have  $score(c_2 \rightarrow p_2) < score(c_1 \rightarrow p_1)$ , 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 naive 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 the a JTT as  $T$ , which consists of tuples  $t_1, \dots, t_m$ . 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.

LEMMA 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.

THEOREM 1.

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

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 5.* 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:  $X = C^Q \rightarrow P^Q$  is bounded by  $M \cdot uscore_b(X, Q) \cdot score_c(X, Q)$ , where  $M = \max(uscore_a(c[i+1], p[1]), uscore_a(c[1], p[j+1]))$ .

### 4.2 Skyline Sweeping Algorithm

Based on Theorem 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 6.* Consider the single pipeline algorithm running on the example in Figure 3(a). Assume that the algorithm has processed  $c[1] \dots c[i]$  on non-free tuple set  $C^Q$  and  $p[1] \dots p[j]$  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 *dominate* 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* [4] 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:  $Q.push(\overbrace{(1, 1, \dots, 1)}^m), calc\_uscore(\overbrace{(1, 1, \dots, 1)}^m))$ 
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:      $top-k.insert(r, score(r))$ 
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  $t.i > 1$  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 [25]. For example, in Figure 3(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.

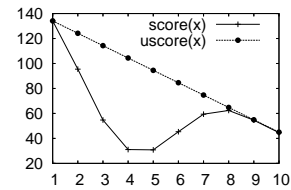
**THEOREM 2.** *The skyline sweeping algorithm has the minimal number of probing to the database.*

#### 4.2.1 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  $Q$ .

### 4.3 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. We illustrate this observation using a one dimensional example in Figure 4. Since the upper bounding function,  $uscore$ , must be monotonic, even if we use more complex functions, it won't be able to approximate the concave part of the real scoring function ( $score$ ) well for  $x$  between 3 to 7.



**Figure 4: Deficiency of Bounding a Non-monotonic Function Using a Monotonic Function**

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 un-processed 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  $Q = \{w_1, \dots, w_m\}$ . 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.,  $\langle tf_{w_1}(t), \dots, tf_{w_m}(t) \rangle$ . 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  $(\sum_{t_i \in T} tf_{w_1}(t_i), \dots, \sum_{t_i \in T} tf_{w_m}(t_i))$ , 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 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:**

$CN$  is the set of CNs.

**Description:**

```

1:  $Q \leftarrow \emptyset$ 
2: for all  $cn \in CN$  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  $top-k[k].score < Q.head().getScore()$  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:    for all 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:      for all 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  $top-k$ 
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 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 7.* Consider the example in Figure 3(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  $c[1], \dots, c[i]$  and  $p[1], \dots, p[j]$  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  $[c[1] \dots c[i]] \times [p[1] \dots p[j]]$ , 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  $bscores$  and  $uscores$  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  $uscores$  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.

## 4.4 Optimizations and Discussions

### 4.4.1 Using Range Parametric Queries

Since our system is external to the RDBMS kernel, executing an SQL query and fetching its results require inter-process communications, and suffer from DBMS internal overheads. If the database is large and the query is complex, a large number of probing queries will be sent to the DBMS. In addition, join selectivity is usually very low and most of such probing queries will return empty result set. Therefore, there will be substantial overhead if we issue a parametric query to check *each* candidate.

In our implementation, we adopt the optimization of grouping candidates into ranges and issue a single parametric query to the database. The selection condition can be rewritten as either a collection of primary key selections obtained from the inverted index combined using `or` or creating a temporary table and using a range selection condition on the row number. Both global pipeline and block pipeline algorithm can benefit from this optimization.

Another benefit of this optimization is that RDBMS might answer such query more efficiently (e.g., by using indexes or avoiding re-accessing/scanning tables multiple times)

### 4.4.2 Instance Optimality

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  $d = \max_{1 \leq i \leq m} d_i$ , 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.

It is our future work to consider an appropriate notion of optimality using a cost model based on the number of database probes.

## 5. 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) <sup>6</sup>, DBLP data (DBLP) [7], and Mondial <sup>7</sup>. All are real datasets. IMDB and DBLP are much larger than Mondial and results on the Mondial dataset are similar to those on IMDB and DBLP, so we will omit Mondial in the rest of the discussion <sup>8</sup>. For the IMDB dataset, we converted a subset of its raw text files into relational tables. Schema and statistics of the two datasets can be found in Tables 4(a) and 4(b).

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. 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 4: Dataset Statistics (Text Attributes Are Underlined)**

(a) IMDB Dataset

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

(b) DBLP Dataset

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

<sup>6</sup><http://www.imdb.com/interfaces>

<sup>7</sup><http://www.dbis.informatik.uni-goettingen.de/Mondial/>

<sup>8</sup>Results on Mondial dataset can be found in the full paper [23].

The databases used are the Linux version of Oracle 10g Express and MySQL v5.0.18, 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 and thus we focus on the results on Oracle. We implemented the **S**parse and global pipeline (**GP**) algorithms, and our skyline sweep (**SS**) and block pipeline algorithms (**BP**). Note that we can lower bound the execution time of the **Hybrid** algorithm [15] as the minimum of the running times of Sparse and GP. We improved the original GP algorithm by using the range parametric query optimization (Section 4.4.1). 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 [22]: (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. We did not use the recall and subsequent MAP measures as we found the recall values are not comparable across different top- $k$  queries.<sup>9</sup> To measure efficiency, we measure the average elapsed times of the algorithms over several runs.

### 5.1 Effectiveness

We show the *reciprocal ranks* of [15], [22], and our proposed method on the DBLP dataset in Table 5. For [22], 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 [15] and [22] methods have a tie in #Rel measure, [22] actually performs better than [15], because it often returns relevant answer(s) within the top-5 results, while [15] 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 [15]’s ranking aggregation method and extremely harsh penalty on the CN sizes, it tends to return results that have only partial matches to

<sup>9</sup>For example, it is not hard to construct a query that will return a large number ( $N$ ) of papers written by a particular author and published in a particular conference. Even the ideal algorithm can only achieve a recall of  $\frac{k}{N}$ , which could be arbitrarily small.

the query or small-sized results. [22] 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 6.

**Table 5: Effectiveness on the DBLP Dataset Based on Top-20 Results**

	[15]	[22]	$p = 1.0$	$p = 1.4$	$p = 2.0$
#Rel	2	2	16	16	18
R-Rank	$\leq 0.243$	$\leq 0.333$	0.926	0.935	1

**Table 7:  $p$ '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 7. 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.

## 5.2 Efficiency

We show running time for all queries on the DBLP and IMDB datasets in Figures 5(a) to 5(d) for  $k = 10$  and  $k = 1$ . Note that the y-axis is in logarithm scale. We can make the following observations:

- BP is usually the fastest 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.
- 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 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 5(e)

to 5(g). 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 GP, SS and BP all perform significantly better than Sparse. There are also queries where GP performs pretty well for small  $k$  values but jumps to a large running cost afterwards (e.g., DQ15). In contrast, although both SS and BP also exhibit a similar increase in query time, they still outperform Sparse. There are also *hard queries* where neither Sparse nor GP runs well (e.g., DQ13). SS still delivers an acceptable performance for some small to medium  $k$  values, and BP can still maintain outstanding performance.

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 5(h). It is obvious that similar conclusions can be drawn about the relative performance of the algorithms.

The fundamental reason of superior performances of SS and BP 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 5(i) and 5(j), 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 5(i) 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. 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 and BP. The reason is that both GP and BP 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 5(k). 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. The DBMS query time ratios of Sparse and SS come as second for different reasons. 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. 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.

## 6. RELATED WORKS

**Keyword Search Systems.** There has been several existing work supporting free-style keyword search on database

Table 6: Top-1 Result for DQ1 (nikos clique) on DBLP

Method	Size	Top-1 Result
[15]	1	<b>InProceeding:</b> <u>Clique-to-Clique</u> Distance Computation Using a Specific Architecture
[22]	6	<b>Person:</b> <u>Nikos Karatzas</u> ← <b>Proceeding</b> → <b>Series</b> ← <b>Proceeding</b> ← <b>InProceeding:</b> Maximum <u>Clique</u> Transversals ← <b>InProceeding:</b> On ... <u>Clique-Width</u> and ...
Ours	3	<b>Person:</b> <u>Nikos Mamoulis</u> ← <b>RPI</b> → <b>InProceeding:</b> Constraint-Based Algorithms for Computing <u>Clique</u> Intersection Joins

contents. Early work includes [12, 13] where entities “near” the occurrence of search keywords are ranked and returned. The idea has been extended to find virtual entities consisting of inter-connected tuples that collectively contain *all* the keywords [1, 16, 3, 18]. Recent work focuses on brining more effective ranking from IR literatures and its related query processing methods [15, 22]. Specifically, [22] improves the ranking method in [15] by the following normalizations: tuple tree size normalization, refined document length normalization, document frequency normalization, and inter-document weight normalization. Some advanced features, such as schema term awareness and phrase-based ranking, are also proposed. Our work can be viewed as a further improvement along the line of enhancing the retrieval effectiveness. [30] is the first to propose a materialization-based approach to answer keyword queries on top of RDBMSs. Most recently, keyword search has been studied under a few generalized contexts too [29, 21].

**Top-*k* Query Processing.** Top-*k* query processing has also been extensively studied in the literature. Fagin *et al.* [10, 11] 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 [24], and minimizing the IO cost [2]. Some other approaches to attack the problem include building indexes [32] or building materialized views [8].

A generalized version of Fagin’s top-*K* problem is to consider complex relationships among objects. [25] studied finding top-*K* joined objects and proposed a  $J^*$  algorithm. A number of improvements were suggested by Iiyas *et al.* [17]. Chang *et al.* considered more general predicates, and an optimal algorithm, MPro, was proposed based on the *necessary probing principle* [5]. However, all the above work assumes a monotonic aggregation function to rank the objects, while our ranking function is non-monotonic and existing methods cannot be immediately applied.

## 7. 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. Two algorithms were proposed that aggressively minimize database probes. 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.

## Acknowledgement

We would like to thank Nino Svonja for his contribution in the early stage of this work. We also want to thank the anonymous reviewers who provided helpful suggestions to the paper.

## 8. REFERENCES

- [1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.
- [2] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.
- [3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.
- [4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
- [5] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.
- [6] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating db and ir technologies: What is the sound of one hand clapping? In *CIDR*, pages 1–12, 2005.
- [7] R. Cyganiak. D2RQ benchmarking. <http://sites.wiwiw.de/fu-berlin.de/suhl/bizer/d2rq/benchmarks/>.
- [8] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.
- [9] 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.
- [10] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.
- [11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [12] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB*, 1998.
- [13] T. Grabs, K. Böhm, and H.-J. Schek. Powerdb-ir - information retrieval on top of a database cluster. In *CIKM*, pages 411–418, 2001.
- [14] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999*, pages 287–298, 1999.
- [15] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, 2003.
- [16] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.
- [17] I. F. Iiyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.
- [18] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.
- [19] B. Kimelfeld and Y. Sagiv. Efficient engines for keyword proximity search. In *WebDB*, pages 67–72, 2005.
- [20] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.
- [21] G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. In *ICDE*, 2006.
- [22] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.
- [23] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: Top-k keyword query in relational databases. Technical Report 0708, School of Computer Science and Engineering, University of New South Wales, 2007.
- [24] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72,

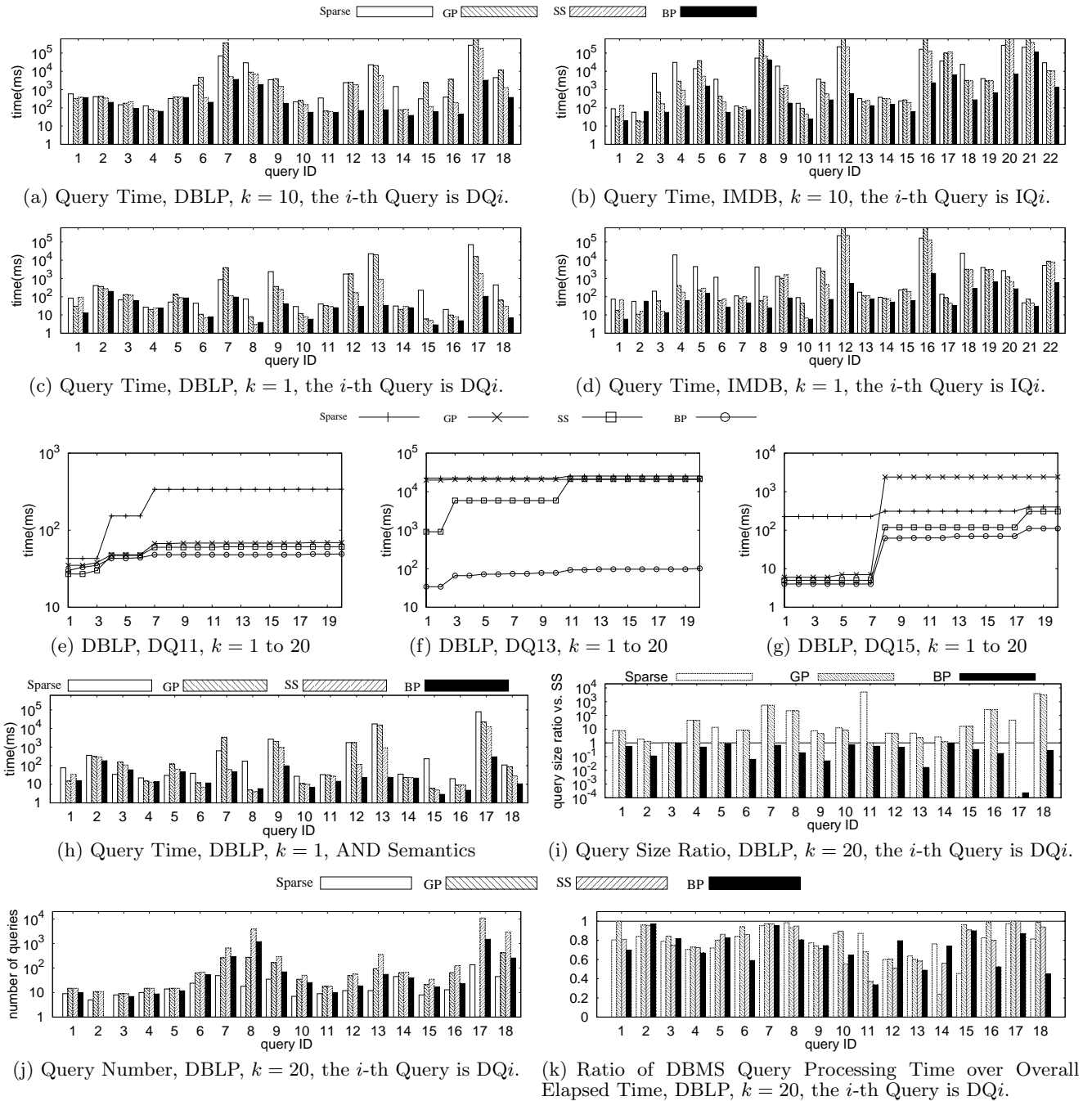


Figure 5: Evaluation of Query Processing Performance

- 2006.
- [25] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.
- [26] S. E. Robertson, H. Zaragoza, and M. J. Taylor. Simple bm25 extension to multiple weighted fields. In *CIKM*, pages 42–49, 2004.
- [27] D. E. Rose and D. R. Cutting. Ranking for usability: Enhanced retrieval for short queries. Technical Report 163, Apple Technical Report, 1996.
- [28] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Communication of the ACM*, 26(11):1022–1036, 1983.
- [29] M. Sayyadan, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, 2007.
- [30] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS*, 2005.
- [31] R. Wilkinson, J. Zobel, and R. Sacks-Davis. Similarity measures for short queries. In *TREC*, 1995.
- [32] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *VLDB*, pages 235–246, 2006.