# Efficient IR-Style Keyword Search over Relational Databases[*]

Vagelis Hristidis
UC, San Diego
vagelis@cs.ucsd.edu

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Yannis Papakonstantinou
UC, San Diego
yannis@cs.ucsd.edu

## Abstract

Applications in which plain text coexists with structured data are pervasive. Commercial relational database management systems (RDBMSs) generally provide querying capabilities for text attributes that incorporate state-of-the-art information retrieval (IR) relevance ranking strategies, but this search functionality requires that queries specify the exact column or columns against which a given list of keywords is to be matched. This requirement can be cumbersome and inflexible from a user perspective: good answers to a keyword query might need to be "assembled" –in perhaps unforeseen ways– by joining tuples from multiple relations. This observation has motivated recent research on free-form keyword search over RDBMSs. In this paper, we adapt IR-style document-relevance ranking strategies to the problem of processing free-form keyword queries over RDBMSs. Our query model can handle queries with both AND and OR semantics, and exploits the sophisticated single-column text-search functionality often available in commercial RDBMSs. We develop query-processing strategies that build on a crucial characteristic of IR-style keyword search: only the few most relevant matches –according to some definition of "relevance"– are generally of interest. Consequently, rather than computing all matches for a keyword query, which leads to inefficient executions, our techniques focus on the top-$k$ matches for the query, for moderate values of $k$. A thorough experimental evaluation over real data shows the performance advantages of our approach.

## 1 Introduction

Applications in which plain text coexists with structured data are pervasive. Furthermore, text and structured data are often stored side by side within standard relational database management systems (RDBMSs), as the following example illustrates.

**Example 1** *Consider a customer-service database from a large vendor of computer equipment. One table in the database, Complaints(prodId, custId, date, comments), logs each complaint received as a tuple with an internal identifier of the customer who made the complaint (custId), an identifier of the main product involved in the complaint (prodId), when the complaint was made (date), and a free-text description of the problem reported by the customer (comments). An example instance of this relation is:*

| *prodId* | *custId* | *date* | *comments* |
|---|---|---|---|
| *p121* | *c3232* | *6-30-2002* | *"disk crashed after just one week of moderate use on an IBM Netvista X41"* |
| *p131* | *c3131* | *7-3-2002* | *"lower-end IBM Netvista caught fire, starting apparently with disk"* |

*The first tuple in this instance corresponds to a complaint by customer c3232 about product p121, which, as we will see, corresponds to a hard drive, on June 30, 2002.*

Commercial RDBMSs generally provide querying capabilities for text attributes that incorporate state-of-the-art information retrieval (IR) relevance ranking strategies. This search functionality requires that queries specify the exact column or columns against which a given list of keywords is to be matched. For example, a query:

```
SELECT * FROM Complaints C
WHERE CONTAINS (C.comments, 'disk crash', 1) > 0
ORDER BY score(1) DESC
```

on Oracle 9.1 [1] returns the rows of the *Complaints* table above that match the keyword query [disk crash], sorted by their *score* as determined by an IR relevance-ranking algorithm. Intuitively, the score of a tuple measures how well its *comments* field matches the query [disk crash].

The requirement that queries specify the exact columns to match can be cumbersome and inflexible from a user perspective: good answers to a keyword query might need to be "assembled" –in perhaps unforeseen ways– by joining tuples from multiple relations:

[*] Work supported by NSF Grant No. 9734548.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

[1] http://www.oracle.com.

**Example 1** *(cont.) An additional relation in our example database, Products(prodId, manufacturer, model), records the manufacturer and model associated with each product. The prodId attribute of the Complaints relation is a foreign-key into Products. Consider the instance of relation Complaints above, plus the following instance of the Products relation:*

| *prodId* | *manufacturer* | *model* |
|---|---|---|
| *p121* | *"Maxtor"* | *"D540X"* |
| *p131* | *"IBM"* | *"Netvista"* |

*Then the best answer for a keyword query [maxtor on ibm netvista] is the tuple that results from joining the first tuple in both relations on the prodId attribute. This join correctly identifies that the complaint by customer c3232 is about a Maxtor disk drive (from the manufacturer attribute of the Products relation) on an IBM Netvista computer (from the comments attribute of the Complaints relation).*

Free-form keyword search over RDBMSs has attracted recent research interest. Given a keyword query, systems such as DBXplorer [1] and DISCOVER [11] join tuples from multiple relations in the database to identify *tuple trees* with all the query keywords ("AND" semantics). All such tuple trees are the answer to the query. Also, both DBXplorer and DISCOVER rank the tuple trees solely by the number of joins needed in their creation. The rationale behind this simple relevance-ranking scheme is that the more joins are needed to create a tuple tree with all query keywords, the less clear it becomes whether the result might be meaningful or helpful. Unfortunately, these techniques do not consider IR-style ranking heuristics that have proved effective over text.

A key contribution of this paper is the incorporation of IR-style relevance ranking of tuple trees into our query processing framework. In particular, our scheme fully exploits single-attribute relevance-ranking results if the RDBMS of choice has text-indexing capabilities (e.g., as is the case for Oracle 9.1, as discussed above). By leveraging state-of-the-art IR relevance-ranking functionality already present in modern RDBMSs, we are able to produce high quality results for free-form keyword queries. For example, a query [disk crash on a netvista] would still match the *comments* attribute of the first *Complaints* tuple above with a high relevance score, after word stemming (so that "crash" matches "crashed") and stop-word elimination (so that the absence of "a" is not weighed too highly). Our scheme relies on the IR engines of RDBMSs to perform such relevance-ranking at the attribute level, and handles both AND and OR semantics.

Unfortunately, existing query-processing strategies for keyword search over RDBMSs are inherently inefficient, since they attempt to capture all tuple trees with all query keywords. Thus these strategies do not exploit a crucial characteristic of IR-style keyword search, namely that only the top 10 or 20 most relevant matches for a keyword query –according to some definition of "relevance"– are generally of interest. The second contribution of this paper is the presentation of efficient query processing techniques for our IR-style queries over RDBMSs that heavily exploit this observation. As we will see, our techniques produce the top-$k$ matches for a query –for moderate values of $k$– in a fraction of the time taken by state-of-the-art strategies to compute all query matches. Furthermore, our techniques are *pipelined*, in the sense that execution can efficiently resume to compute the "next-$k$" matches if the user so desires.

The rest of the paper is structured as follows: Section 2 discusses related work. Then, Sections 3 and 4 define the problem of processing keyword-search top-$k$ queries over RDBMSs, provide necessary notation, and describe the general architecture of the system. Section 5 introduces the key query processing algorithms, which we evaluate experimentally in Section 6. Finally, Section 7 concludes the paper.

## 2 Related Work

Recent research has addressed the problem of free-form keyword search over structured and semi-structured data. BANKS [2] views a database as a graph where the database tuples (or objects) are the nodes and application-specific "relationships" are the edges. For example, an edge may denote a foreign-key relationship. BANKS answers keyword queries by searching for Steiner trees [15] containing all keywords, using heuristics during the search. Goldman et al. [8] use a related graph-based view of databases. A user query specifies two sets of objects, the "*Find*" and the "*Near*" objects, which may be generated using two separate keyword sets. The system then ranks the objects in *Find* according to their distance from the objects in *Near*, using an algorithm that efficiently calculates these distances by building "hub indexes." A drawback of these approaches is that a graph of the database tuples must be materialized and maintained. Furthermore, the important structural information provided by the database schema is ignored, once the data graph has been built.

DBXplorer [1] and DISCOVER [11] exploit the RDBMS schema, which leads to relatively efficient algorithms for answering keyword queries because the structural constraints expressed in the schema are helpful for query processing. These two systems rely on a similar architecture, on which we also build in this paper (Section 4). Unlike DBXplorer and DISCOVER, our techniques are not limited to Boolean-AND semantics for queries, and we can handle queries with both AND and OR semantics. In contrast, DBXplorer and DISCOVER (as well as BANKS) require that all query keywords appear in the tree of nodes or tuples that are returned as the answer to a query. Furthermore, we employ ranking techniques developed by the IR community, instead of ranking answers solely based on the size of the result as in DBXplorer and DISCOVER. Also, our techniques improve on previous work in terms of efficiency by exploiting the fact that free-form keyword queries can generally be answered with just the few most relevant matches. Our work then produces the "top-$k$" matches for a query fast, for moderate values of $k$.

The IR community has focused over the last few decades on improving the quality of relevance-ranking functions for text document collections [16]. We refer the reader to [17] for a recent survey. Our proposed query-processing system builds on the IR work by exploiting the IR-style relevance-ranking functionality that modern RDBMSs routinely include, typically over individual text attributes. For example, Oracle 9i Text[2] and IBM DB2 Text Information Extender[3] use standard SQL to create full text indexes on text attributes of relations. Microsoft SQL Server 2000[4] also provides tools to generate full text indexes, which are stored as files outside the database. All these systems allow users to create full-text indexes on single attributes to then perform keyword queries. By treating these single-attribute indexing modules as "black boxes," our query processing system separates itself from the peculiarities of each attribute domain or application. In effect, our approach does not require any semantic knowledge about the database, and cleanly separates the relevance-ranking problem for a specific database attribute –which is performed by appropriate RDBMS modules– from the problem of combining the individual attribute scores and identifying the top "joining trees of tuples" (see Section 3) for a query, which becomes the focus of our work.

Keyword search over XML databases has also attracted interest recently [7, 12, 9]. Florescu et al. [7] extend XML query languages to enable keyword search at the granularity of XML elements, which helps novice users formulate queries. This work does not consider keyword proximity. Hristidis et al. [12] view an XML database as a graph of "minimal" XML segments and find connections between them that contain all the query keywords. They focus on the presentation of the results and use view materialization techniques to provide fast response times. Finally, XRANK [9] proposes a ranking function for the XML "result trees", which combines the scores of the individual nodes of the result tree. The tree nodes are assigned PageRank-style scores [3] off-line. These scores are query independent and, unlike our work, do not incorporate IR-style keyword relevance.

The problem of processing "top-$k$" queries has attracted recent attention in a number of different scenarios. The design of the pipelined algorithms that we propose in this paper faces challenges that are related to other top-$k$ work (e.g., [14, 6, 10, 4]). However, our problem is unique (Section 5) in that we need to join (ranked) tuples coming from multiple relations in unpredictable ways to produce the final top-$k$ results.

Finally, Natsev et al. [13] extend the work by Fagin et al. [6] by allowing different objects to appear in the source "lists," as opposed to assuming that the lists have just attribute values for a common set of objects. As a result, the objects from the lists need to be joined, which is done via user-defined aggregation functions. The *Single Pipelined* algorithm of Section 5.3 can be regarded as an instance of the more general $J*$ algorithm by Natsev et al. [13]. However, $J*$ does not consider predicates over "connecting" relations (i.e., free tuple sets in the terminology of Section 4). Also, during processing $J*$ buffers all incomplete results, which would be inefficient (or even infeasible) for our setting, where all combinations of tuples from the non-free tuple sets are candidate results (i.e., may join through the free tuple sets).

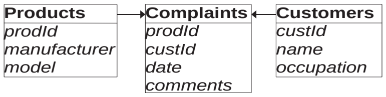

Figure 1: Schema of the *Complaints* database.

| Complaints | | | | |
|---|---|---|---|---|
| *tupleId* | *prodId* | *custId* | *date* | *comments* |
| $c_1$ | p121 | c3232 | 6-30-2002 | "disk crashed after just one week of moderate use on an IBM Netvista X41" |
| $c_2$ | p131 | c3131 | 7-3-2002 | "lower-end IBM Netvista caught fire, starting apparently with disk" |
| $c_3$ | p131 | c3143 | 8-3-2002 | "IBM Netvista unstable with Maxtor HD" |

| Products | | | |
|---|---|---|---|
| *tupleId* | *prodId* | *manufacturer* | *model* |
| $p_1$ | p121 | "Maxtor" | "D540X" |
| $p_2$ | p131 | "IBM" | "Netvista" |
| $p_3$ | p141 | "Tripplite" | "Smart 700VA" |

| Customers | | | |
|---|---|---|---|
| *tupleId* | *custId* | *name* | *occupation* |
| $u_1$ | c3232 | "John Smith" | "Software Engineer" |
| $u_2$ | c3131 | "Jack Lucas" | "Architect" |
| $u_3$ | c3143 | "John Mayer" | "Student" |

Figure 2: An instance of the *Complaints* database.

## 3 Framework

In this section, we specify the query model (Section 3.1), together with the family of scoring functions that we consider to identify the top-$k$ answers for a query (Section 3.2).

### 3.1 Query Model

Consider a database with $n$ relations $R_1, \ldots, R_n$. Each relation $R_i$ has $m_i$ attributes $a^i_1, \ldots, a^i_{m_i}$, a primary key and possibly foreign keys into other relations. The *schema graph* $G$ is a directed graph that captures the foreign key relationships in the schema. $G$ has a node for each relation $R_i$, and an edge $R_i \rightarrow R_j$ for each primary key to foreign key relationship from $R_i$ into $R_j$. Figure 1 shows the schema graph of our *Complaints* database running example, while Figure 2 shows a possible instance of this database. We use schema graphs in the following definition, which forms the basis for the query-result specification:

**Definition 1 (Joining trees of tuples)** *Given a schema graph $G$ for a database, a joining tree of tuples $T$ is a tree of tuples where each edge $(t_i, t_j)$ in $T$, where $t_i \in R_i$ and $t_j \in R_j$, satisfies two properties: (1) $(R_i, R_j) \in G$, and (2) $t_i \bowtie t_j \in R_i \bowtie R_j$. The $size(T)$ of a joining tree $T$ is the number of tuples in $T$.*

[2] http://technet.oracle.com/products/text/content.html.
[3] http://www.ibm.com/software/data/db2/extenders/textinformation/.
[4] http://msdn.microsoft.com/library/.

A *top-$k$ keyword query* is a list of keywords $Q = [w_1, \ldots, w_m]$. The result for such a top-$k$ query is a list of the $k$ joining trees of tuples $T$ whose $Score(T, Q)$ score for the query is highest, where $Score(T, Q)$ is discussed below. (Ties are broken arbitrarily.) The query result is sorted in descending order of the scores. We require that any joining tree $T$ in a query result be minimal: if a tuple $t$ with zero score is removed from $T$, then the tuples remaining in $T$ are "disconnected" and do not form a joining tree. In other words, $T$ cannot have a leaf tuple with zero score. As an example, for a choice of ranking function $Score$ the results for a top-3 query [Netvista Maxtor] over our *Complaints* database could be (1) $c_3$; (2) $p_2 \rightarrow c_3$; and (3) $p_1 \rightarrow c_1$. Finally, we do not allow any tuple to appear more than once in a joining tree of tuples.

### 3.2 Ranking Functions

We now discuss how to rank joining trees of tuples for a given query. Result ranking has been addressed by other keyword-search systems for relational data. Given a query $Q$, both DISCOVER [11] and DBXplorer [1] assign a score to a joining tree of tuples $T$ in the following way:

$$Score(T, Q) = \begin{cases} \frac{1}{size(T)} & \text{if } T \text{ contains all words in } Q \\ 0 & \text{otherwise} \end{cases}$$

Alternatively, BANKS [2] uses the following scoring scheme:[5]

$$Score(T, Q) = \begin{cases} f_r(T) + f_n(T) + f_p(T) & \text{if } T \text{ contains} \\ & \text{all words in } Q \\ 0 & \text{otherwise} \end{cases}$$

where $f_r(T)$ measures how "related" the relations of the tuples of $T$ are, $f_n(T)$ depends on the weight of the tuples of $T$ –as determined by a PageRank-inspired technique–, and $f_p(T)$ is a function of the weight of the edges of $T$.

The approaches above capture the size and "structure" of a query result in the score that it is assigned, but do not leverage further the relevance-ranking strategies developed by the IR community over the years. As discussed in the introduction, these strategies –which were developed exactly to improve document-ranking quality for free-form keyword queries– can naturally help improve the quality of keyword query results over RDBMSs. Furthermore, modern RDBMSs already include IR-style relevance ranking functionality over individual text attributes, which we exploit to define our ranking scheme. Specifically, the score that we assign to a joining tree of tuples $T$ for a query $Q$ relies on:

- Single-attribute IR-style relevance scores $Score(a_i, Q)$ for each textual attribute $a_i \in T$ and query $Q$, as determined by an IR engine at the RDBMS, and
- A function $Combine$, which combines the single-attribute scores into a final score for $T$.

As an example, a state-of-the-art IR definition for a single-attribute scoring function $Score$ is as follows [17]:

$$Score(a_i, Q) = \sum_{w \in Q \cap a_i} \frac{1 + ln(1 + ln(tf))}{(1 - s) + s\frac{dl}{avdl}} \cdot ln\frac{N + 1}{df} \quad (1)$$

where, for a word $w$, $tf$ is the frequency of $w$ in $a_i$, $df$ is the number of tuples in $a_i$'s relation with word $w$ in this attribute, $dl$ is the size of $a_i$ in characters, $avdl$ is the average attribute-value size, $N$ is the total number of tuples in $a_i$'s relation, and $s$ is a constant (usually 0.2). (Note that this single-attribute scoring function can be easily extended to incorporate PageRank-style "link"-based scores [3, 9].)

We now turn to the problem of combining single-attribute scores for a joining tree of tuples $T$ into a final score for the tree. Notice that the score for a single tuple $t$ is defined by viewing $t$ as a joining tree of size 1. Let $A = \langle a_1, \ldots, a_n \rangle$ be a vector with all textual attribute values for $T$. We define the score of $T$ for $Q$ as $Score(T, Q) = Combine(\mathbf{Score}(A, Q),\ size(T))$, where $\mathbf{Score}(A, Q) = \langle Score(a_1, Q), \ldots, Score(a_n, Q) \rangle$. (Notice that instead of $size(T)$ we could use other characteristics of $T$, as suited to the specifics of the application.) A simple definition for $Combine$ is:

$$Combine(\mathbf{Score}(A, Q), size(T)) = \frac{\sum_{a_i \in A} Score(a_i, Q)}{size(T)} \quad (2)$$

The definition for the $Combine$ function above is a natural one, but of course other such functions are possible. The query processing algorithms that we present later can handle any combining function that satisfies the following property:

**Definition 2 (Tuple monotonicity)** *A combining function $Combine$ satisfies the* tuple monotonicity *property if, for every query $Q$ and joining trees of tuples $T$ and $T'$ derived from the same CN such that (i) $T$ consists of tuples $t_1, \ldots, t_n$ while $T'$ consists of tuples $t'_1, \ldots, t'_n$ and (ii) $Score(t_i, Q) \leq Score(t'_i, Q)$ for all $i$, it follows that $Score(T, Q) \leq Score(T', Q)$.*

Notice that the ranking function $Score(t, Q)$ for a single tuple can be arbitrary, although in the above discussion we assume that the same formula (Equation 2) calculates the rank for both a single tuple and a joining tree of tuples. All ranking functions for joining trees of tuples of which we are aware [1, 11, 2], including the one in Equation 2, satisfy the tuple-monotonicity property, and hence can be used with the execution algorithms that we discuss later in this paper.

In addition to the combining function, queries should specify whether they have *Boolean AND or OR semantics*. The AND semantics assigns a score of 0 to any tuple tree that does not include all query keywords, while tuple trees with all query keywords receive the score determined by $Combine$. In contrast, the OR semantics always assigns a tuple tree its score as determined by $Combine$, whether the tuple tree includes all query keywords or not.

In summary, the single-attribute $Score$ function, together with the $Combine$ function of choice, assign relevance scores to joining trees of tuples either with AND or with OR semantics. The next section outlines the architecture of our query processing system, which efficiently identifies the trees of tuples with the highest relevance scores for a given query.

[5] Reference [2] introduces several variations of this scheme (e.g., the tuple and edge terms above could be multiplied rather than added).

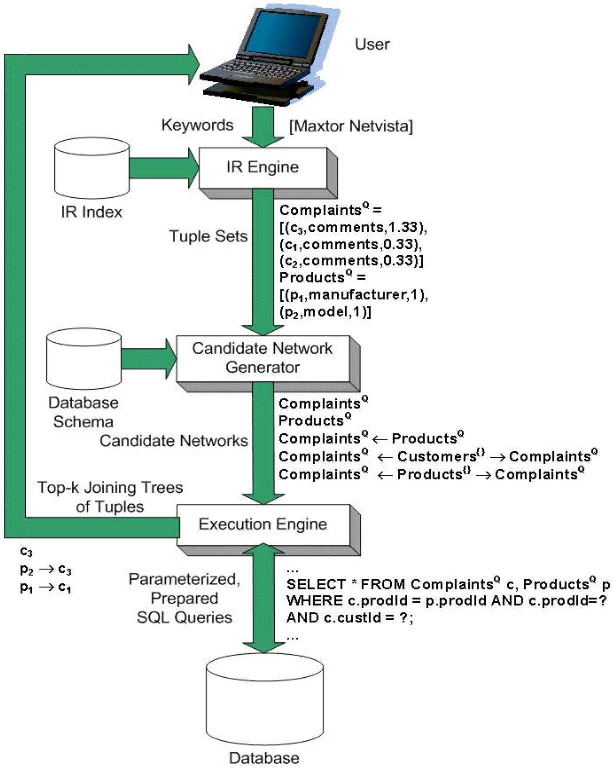

Figure 3: Architecture of our query processing system.

## 4 System Architecture

The architecture of our query processing system relies whenever possible on existing, unmodified RDBMS components. Specifically, our architecture (Figure 3) consists of the following modules:

### 4.1 IR Engine

As discussed, modern RDBMSs include IR-style text indexing functionality at the attribute level. The *IR Engine* module of our architecture exploits this functionality to identify all database tuples that have a non-zero score for a given query. The *IR Engine* relies on the *IR Index*, which is an inverted index that associates each keyword that appears in the database with a list of occurrences of the keyword; each occurrence of a keyword is recorded as a tuple-attribute pair. Our implementation uses Oracle Text, which keeps a separate index for each relation attribute. We combine these individual indexes to build the *IR Index*.[6]

When a query $Q$ arrives, the *IR Engine* uses the *IR Index* to extract from each relation $R$ the *tuple set* $R^Q = \{t \in R \mid Score(t, Q) > 0\}$, which consists of the tuples of $R$ with a non-zero score for $Q$. The tuples $t$ in the tuple sets are ranked in descending order of $Score(t, Q)$, as required by the top-$k$ query processing algorithms described below.

### 4.2 Candidate Network Generator

The next module in the pipeline is the *Candidate Network (CN) Generator*, which receives as input the non-empty tuple sets from the *IR Engine*, together with the database schema and a parameter $M$ that we explain below. The key role of this module is to produce CNs, which are join expressions to be used to create joining trees of tuples that will be considered as potential answers to the query.

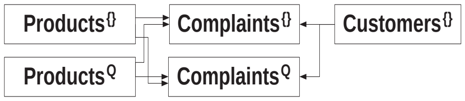

Figure 4: Tuple set graph for the *Complaints* database and query [Maxtor Netvista].

Specifically, a CN is a join expression that involves tuple sets plus perhaps additional "base" database relations. We refer to a base relation $R$ that appears in a CN as a *free tuple set* and denote it as $R^{\{\}}$. Intuitively, the free tuple sets in a CN do not have occurrences of the query keywords, but help "connect" (via foreign-key joins) the (non-free) tuple sets that do have non-zero scores for the query. Each result $T$ of a CN is thus a potential result of the keyword query. We say that a joining tree of tuples $T$ *belongs* to a CN $C$ ($T \in C$) if there is a tree isomorphism mapping $h$ from the tuples of $T$ to the tuple sets of $C$. For example, in Figure 2, $(c_1 \leftarrow p_1) \in (Complaints^Q \leftarrow Products^Q)$. The input parameter $M$ bounds the size (in number of tuple sets, free or non-free) of the CNs that this module produces.

The notion of CN was introduced in DBXplorer [1][7] and DISCOVER [11]. As discussed, DISCOVER and DBXplorer require that each joining tree of tuples in the query answer contain all query keywords. To produce all answers for a query with this AND semantics, these systems create *multiple* tuple sets for each database relation. Specifically, a separate tuple set is created for each combination of keywords in $Q$ and each relation. This generally leads to a number of CNs that is exponential in the query size, which makes query execution prohibitively expensive for queries of more than a very small number of keywords or for values of $M$ greater than 4 or so.

In contrast, we only create a single tuple set $R^Q$ for each relation $R$, as specified above. For queries with AND semantics, a postprocessing step checks that we only return tuple trees containing all query keywords. As we will see, this characteristic of our system results in significantly faster executions, which in turn allows us to handle larger queries and also consider larger CNs.

The CN generation algorithm is based on that of the DISCOVER system, and is not explained here in full detail due to lack of space. Conceptually, we first create the *tuple set graph* from the database schema graph and the tuple sets returned by the *IR Engine* module. Figure 4 shows the tuple set graph for the *Complaints* database and query $Q =$ [Maxtor Netvista]. Initially, the set $S$ of candidate CNs consists of the non-free tuple sets ($Products^Q$ and $Complaints^Q$ in our example). We progressively expand each CN $s \in S$ by adding a tuple set adjacent to $s$ in the tuple set graph. We consider $s$ to be a CN and hence part of

[6] In principle, we could exploit more efficient indexing schemes (e.g., text indexes at the tuple level) as RDBMSs start to support them.

[7] DBXplorer refers to CNs as "join trees."

| $C^Q$ | $P^Q$ | $C^Q \leftarrow P^Q$ | $C^Q \leftarrow U^{\{\}} \rightarrow C^Q$ | $C^Q \leftarrow P^{\{\}} \rightarrow C^Q$ |
|---|---|---|---|---|
| $c_3$: 1.33<br>$c_1$: 0.33<br>$c_2$: 0.33 | $p_1$: 1<br>$p_2$: 1 | $c_3 \leftarrow p_2$: 1.17<br>$c_1 \leftarrow p_1$: 0.67<br>$c_2 \leftarrow p_2$: 0.67 | | $c_3 \leftarrow p_2 \rightarrow c_2$: 1.11 |

Figure 5: CN results for the *Complaints* database and query [Maxtor Netvista], where $C$ stands for *Complaints*, $P$ for *Products*, and $U$ for *Customers*.

the output of this module if it satisfies the following properties:

1. *The number of non-free tuple sets in $s$ does not exceed the number of query keywords $m$:* This constraint guarantees that we generate a minimum number of CNs while not missing any result that contains all the keywords, which is crucial for Boolean-AND semantics. That is, for every result $T$ that contains every keyword exactly once, a CN $C$ exists such that $T \in C$. For example, $Products^Q \rightarrow Complaints^Q \leftarrow Customers^{\{\}} \rightarrow Complaints^Q$ is not a CN of query [Maxtor Netvista]. In particular, its results are a subset of the results of $Products^Q \rightarrow Complaints^{\{\}} \leftarrow Customers^{\{\}} \rightarrow Complaints^Q$.
2. *No leaf tuple sets of $s$ are free:* This constraint ensures CN "minimality." For example, $Products^Q \rightarrow Complaints^{\{\}}$ is not a CN because it is subsumed by the simpler CN $Products^Q$.
3. *$s$ does not contain a construct of the form $R \rightarrow S \leftarrow R$:* If such a construct existed, every resulting joining tree of tuples would contain the same tuple more than once. For example, $Products^Q \rightarrow Complaints^{\{\}} \leftarrow Products^Q$ is not a CN because all produced joining networks of tuples would be of the form $p \rightarrow c \leftarrow p'$ with $p \equiv p'$.

The size of a CN is its number of tuple sets. All CNs of size 3 or lower for the query [Maxtor Netvista] are shown in Figure 3.

### 4.3 Execution Engine

The final module in the pipeline is the *Execution Engine*, which receives as input a set of CNs together with the non-free tuple sets. The *Execution Engine* contacts the RDBMS's query execution engine repeatedly to identify the top-$k$ query results. Figure 5 shows the joining trees of tuples produced by each CN, together with their scores for the query [Maxtor Netvista] over our *Complaints* example. The *Execution Engine* module is the most challenging to implement efficiently, and is the subject of the next section.

## 5 Execution Algorithms

We now present algorithms for a core operation in our system: given a set of CNs together with a set of non-free tuple sets, the *Execution Engine* needs to efficiently identify the top-$k$ joining trees of tuples that can be derived. First, we describe the *Naive* algorithm, a simple adaptation of query processing algorithms used in prior work [11, 1]. Second, we present the *Sparse* algorithm, which improves on the *Naive* algorithm by dynamically pruning some CNs during query evaluation. Third, we describe the *Single Pipelined* algorithm, which calculates the top-$k$ results for a single CN in a pipelined way. Fourth, we present the *Global Pipelined* algorithm, which generalizes the *Single Pipelined* algorithm to multiple CNs and can then be used to calculate the final result for top-$k$ queries. Finally, we introduce the *Hybrid* algorithm, which combines the virtues of both the *Global Pipelined* and the *Sparse* algorithms, and is shown to outperform all other approaches in Section 6.

### 5.1 Naive Algorithm

The *Naive* algorithm issues a SQL query for each CN for a top-$k$ query. The results from each CN are combined in a sort-merge manner to identify the final top-$k$ results of the query. This approach is an adaptation of the execution algorithms of prior work [11, 1, 12] for keyword-search queries. As a simple optimization in our experiments, we only get the top-$k$ results from each CN according to the scoring function, and we enable the top-$k$ "hint" functionality, available in the Oracle 9.1 RDBMS.[8] In the case of Boolean-AND semantics, the *Naive* algorithm (as well as the *Sparse* algorithm presented below) involves an additional filtering step on the stream of results to check for the presence of all keywords.

### 5.2 Sparse Algorithm

The *Naive* algorithm exhaustively processes every CN associated with a query. We can improve query-processing performance by discarding at any point in time any (unprocessed) CN that is guaranteed not to produce a top-$k$ match for the query. Specifically, the *Sparse* algorithm computes a bound $MPS_i$ on the maximum possible score of a tuple tree derived from a CN $C_i$. If $MPS_i$ does not exceed the actual score of $k$ already produced tuple trees, then CN $C_i$ can be safely removed from further consideration. To calculate $MPS_i$, we apply the combining function to the top tuples (due to the monotonicity property in Definition 2) of the non-free tuple sets of $C_i$. That is, $MPS_i$ is the score of a hypothetical joining tree of tuples $T$ that contains the top tuples from every non-free tuple set in $C_i$. As a further optimization, the CNs for a query are evaluated in ascending size order. This way, the smallest CNs, which are the least expensive to process and are the most likely to produce high-score tuple trees using the combining function above, are evaluated first. As we discuss in Section 6.3, the *Sparse* algorithm is the method of choice for queries that produce relatively few results.

### 5.3 Single Pipelined Algorithm

The *Single Pipelined* algorithm (Figure 7) receives as input a candidate network $C$ and the non-free tuple sets $TS_1, \ldots, TS_v$ that participate in $C$. Recall that each of these non-free tuple sets corresponds to one relation, and contains the tuples in the relation with a non-zero match

[8] This hint to the optimizer has not significantly improved performance in our experiments.

for the query. Furthermore, the tuples in $TS_i$ are sorted in descending order of their $Score$ for the query. (Note that the attribute $Score(a_i, Q)$ and tuple $Score(t, Q)$ scores associated with each tuple $t \in TS_i$ are initially computed by the *IR Engine*, as we described, and do not need to be re-calculated by the *Execution Engine*.) The output of the *Single Pipelined Algorithm* consists of a stream of joining trees of tuples $T$ in descending $Score(T, Q)$ order.

The intuition behind the *Single Pipelined* algorithm is as follows. We keep track of the prefix $S(TS_i)$ that we have retrieved from every tuple set $TS_i$; in each iteration of the algorithm, we retrieve a new tuple $t$ from one $TS_M$, after which we add it to the associated retrieved prefix $S(TS_M)$. (We discuss the choice of $TS_M$ below.) Then, we proceed to identify each potential joining tree of tuples $T$ in which $t$ can participate. For this, we prepare in advance a *parameterized query* that performs appropriate joins involving the retrieved prefixes. (Figure 3 shows the parameterized query for the CN $C^Q \leftarrow P^Q$.) Specifically, we invoke this parameterized query once for every tuple $(t_1, \ldots, t_{M-1}, t, t_{M+1}, \ldots, t_v)$, where $t_i \in S(TS_i)$ for $i = 1, \ldots, v$ and $i \neq M$. All joining trees of tuples that include $t$ are returned by these queries, and are added to a queue $R$. We cannot output these trees until we can guarantee that they are one of the top-$k$ joining trees for the original query. Notice that a naive execution of this algorithm would prevent us from producing any results until all candidate trees are computed and rank-ordered. As we discuss next, we bound the score that tuple trees not yet produced can achieve, hence circumventing this limitation of naive algorithms.

In effect, the *Single Pipelined* algorithm can start producing results before examining the entire tuple sets. For this, we maintain an effective estimate of the *Maximum Possible Future Score (MPFS)*[9] that any unseen result can achieve, given the information already gathered by the algorithm. Specifically, we analyze the status of each prefix $S(TS_i)$ to bound the maximum score that an unretrieved tuple from the corresponding non-free tuple set can reach. (Recall once again that non-free tuple sets are ordered by their tuple scores.) To compute $MPFS$, we first calculate $MPFS_i$ for each non-free tuple set $TS_i$ as the maximum possible future score of any tuple tree that contains a tuple from $TS_i$ that has not yet been retrieved (i.e., that is not in $S(TS_i)$):

$$MPFS_i = \max\{Score(T, Q) \mid$$
$$T \in TS_1 \bowtie \ldots \bowtie (TS_i - S(TS_i)) \bowtie \ldots \bowtie TS_v\}$$

Unfortunately, a precise calculation of $MPFS_i$ would require multiple database queries, with cost similar to that of computing all possible tuple trees for the queries. As an alternative to this expensive computation, we attempt to produce a (hopefully tight) overestimate $\overline{MPFS_i}$, computed as the score of the hypothetical tree of tuples consisting of the next unprocessed tuple $t_i$ from $TS_i$ and the top-ranked tuple $t_j^{top}$ of each tuple set $TS_j$, for $j \neq i$. Notice that $\overline{MPFS_i}$ is an overestimate of $MPFS_i$ because there is no guarantee that the tuples $t_i$ and $t_j^{top}$ will indeed participate in a joining tree of $C$. However, $\overline{MPFS_i}$ is the best estimate that we can produce efficiently without accessing the database and, as we will see, results in significant savings over the naive executions. Following a similar rationale, we also define an overestimate $\overline{MPFS}$ for the entire candidate network $C$, as $\overline{MPFS} = \max_{i=1,\ldots,v} \overline{MPFS_i}$. A tentative result from $R$ (see Figure 7) is safely returned as one of the top-$k$ results if its associated score is no less than $\overline{MPFS}$.

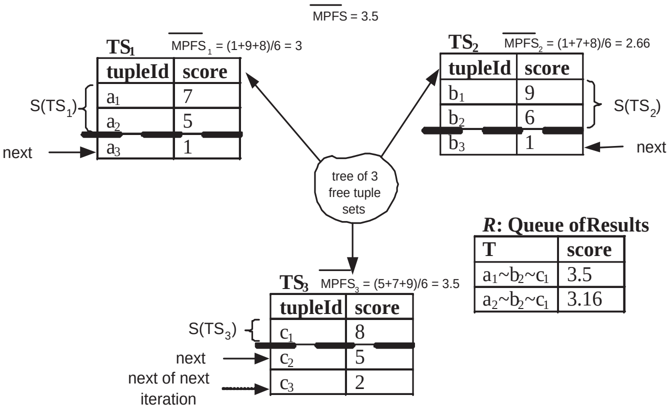

Figure 6: Snapshot of a *Single Pipelined* execution.

Another key issue is the choice of the tuple set from which to pick the next tuple $t$. One simple possibility is to pick tuple sets randomly, or in a round-robin way. Instead, the *Single Pipelined* algorithm picks the "most promising" tuple set, which is defined as the tuple set that can produce the highest ranked result. Using this heuristic, we pick the next tuple from the tuple set $TS_M$ with the maximum value of $\overline{MPFS_i}$ (i.e., $\overline{MPFS_M} = \max_i \overline{MPFS_i}$). The experiments of Section 6 show that this choice of tuple set results in better performance over random or round-robin choices.

**Example 2** *Figure 6 shows a snapshot of an execution of the* Single Pipelined *algorithm on a hypothetical database. The candidate network $C$ has three free and three non-free tuple sets. The thick dotted lines denote the prefix of each tuple set retrieved so far. The combining function of Equation 2 is used. The first result of $R$ is output because its score is equal to $\overline{MPFS}$. In contrast, the second result cannot yet be safely output because its score is below $\overline{MPFS}$. Suppose that we now retrieve a new tuple $c_2$ from the tuple set with maximum $\overline{MPFS_i}$. Further, assume that no results are produced by the associated parameterized queries when instantiated with $c_2$. Then, $\overline{MPFS_3} = \frac{2+7+9}{6} = 3$ and $\overline{MPFS} = 3$. Hence now the second result of $R$ can be output.*

The correctness of this algorithm relies on the combining function satisfying the tuple monotonicity property from Definition 2. Notice that the following extra step is needed for queries with AND semantics: Before issuing a parameterized query, we check if *all* query keywords are contained in the tuples that are passed as parameters. As we will see in Section 6, the *Single Pipelined* algorithm is not an efficient choice when used separately for each CN, but is the main building block of the efficient *Global Pipelined* algorithm described below.

[9] Notice that $MPS$, as defined in Section 5.2, is equivalent to $MPFS$ before the evaluation of the CN begins (i.e., before any parameterized query is executed).

```
Single Pipelined Algorithm($C$, $Q$, $k$, $Score(.)$, $TS_1, \ldots, TS_v$){
$h(TS_i)$: top unprocessed tuple of $TS_i$
        (i.e., not yet added to $S(TS_i)$)
$S(TS_i)$: prefix of $TS_i$ retrieved so far; initially empty
$R$: queue for not-yet-output results, by descending $Score(T, Q)$
Execute parameterized query $q(h(TS_1), \ldots, h(TS_v))$
Add results of $q$ to $R$
Output all results $T$ in $R$ with $Score(T, Q) \geq \max_{i=1}^{v} \overline{MPFS}_i$
For $i = 1, \ldots, v$ move $h(TS_i)$ to $S(TS_i)$
While (fewer than $k$ results have been output) do {
  Get tuple $t = h(TS_M)$, where $\overline{MPFS}_M = \max_{i=1}^{v} \overline{MPFS}_i$
  Move $t$ to $S(TS_M)$
  For each combination $(t_1, \ldots, t_{M-1}, t_{M+1}, \ldots, t_v)$ of tuples
        where $t_i \in S(TS_i)$ do {
    Execute parameterized query $q(t_1, \ldots, t_{M-1}, t, t_{M+1}, \ldots, t_v)$
    Add results of $q$ to $R$}
  Output all new results $T$ in $R$ with
        $Score(T, Q) \geq \max_{i=1}^{v} \overline{MPFS}_i$}}
```

Figure 7: The *Single Pipelined* algorithm.

```
Global Pipelined Algorithm($C_1, \ldots, C_n$, $k$, $Q$, $Score(.)$){
01. Let $v_i$ be the number of non-free tuple sets of CN $C_i$
02. $h(TS_{i,j})$: top unprocessed tuple of $C_i$'s $j$-th tuple set $TS_{i,j}$
03. $S(TS_{i,j})$: prefix of $TS_{i,j}$ retrieved so far; initially empty
04. $R$: queue for not-yet-output results, by descending $Score(T, Q)$
05. For $i = 1 \ldots n$ do {
06.   Execute parameterized query $q_i(h(TS_{i,1}), \ldots, h(TS_{i,v_i}))$
07.    /* $q_i$ is the parameterized query for CN $C_i$ */
08.   Add results of $q_i$ to $R$
09.   For $j = 1, \ldots, v_i$ move $h(TS_{i,j})$ to $S(TS_{i,j})$}
10. Output all results $T$ in $R$ with $Score(T, Q) \geq GMPFS$
11. While (fewer than $k$ results have been output) do {
12.   /* Get tuple from most promising tuple set of most promising CN */
13.   Get tuple $t = h(TS_{c,M})$, where $\overline{MPFS}_M$ for CN $C_c$ is highest
14.   Move $t$ to $S(TS_{c,M})$
15.   For each combination $(t_1, \ldots, t_{M-1}, t_{M+1}, \ldots, t_{v_c})$ of tuples
            where $t_l \in S(TS_{c,l})$ do{
16.     Execute parameterized query $q_c(t_1, \ldots, t_{M-1}, t, t_{M+1}, \ldots, t_{v_c})$
17.     Add results of $q_c$ to $R$}
18.   Output all new results $T$ in $R$ with $Score(T, Q) \geq GMPFS$}}
```

Figure 8: The *Global Pipelined* algorithm.

### 5.4 Global Pipelined Algorithm

The *Global Pipelined* algorithm (Figure 8) builds on the *Single Pipelined* algorithm to efficiently answer a top-$k$ keyword query over multiple CNs. The algorithm receives as input a set of candidate networks, together with their associated non-free tuple sets, and produces as output a stream of joining trees of tuples ranked by their overall score for the query.

The key idea of the algorithm is the following. All CNs of the keyword query are evaluated concurrently following an adaptation of a *priority preemptive, round robin* protocol [5], where the execution of each CN corresponds to a process. Each CN is evaluated using a modification of the *Single Pipelined* algorithm, with the "priority" of a process being the $\overline{MPFS}$ value of its associated CN.

Initially, a "minimal" portion of the most promising CN $C_c$ (i.e., $C_c$ has the highest $\overline{MPFS}$ value) is evaluated. Specifically, this minimal portion corresponds to processing the next tuple from $C_c$ (lines 12–17). After this, the priority of $C_c$ (i.e., $\overline{MPFS}_c$) is updated, and the CN with the next highest $\overline{MPFS}$ value is picked. A tuple-tree result is output (line 18) if its score is no lower than the current value of the *Global* $\overline{MPFS}$, $GMPFS$, defined as the maximum $\overline{MPFS}$ among all the CNs for the query. Note that if the same tuple set $TS$ is in two different CNs, it is processed as two separate (but identical) tuple sets. In practice, this is implemented by maintaining two open cursors for $TS$.

```
Hybrid Algorithm($C_1, \ldots, C_n$, $k$, $c$, $Q$, $Score(.)$){
$c$ is a tuning constant
$Estimate = GetEstimate(C_1, \ldots, C_n)$
If $Estimate > c \cdot k$ then execute Global Pipelined
else execute Sparse}
```

Figure 10: The *Hybrid* algorithm.

**Example 3** *Figure 9 shows a snapshot of the* Global Pipelined *evaluation of a query with five CNs on a hypothetical database. At each point, we process the CN with the maximum* $\overline{MPFS}$*, and maintain a global queue of potential results. After a minimal portion of the current CN* $C$ *is evaluated, its* $\overline{MPFS}$ *is updated, which redefines the priority of* $C$*.*

**Example 4** *Consider query [Maxtor Netvista] on our running example. We consider all CNs of size up to* 2*, namely* $C_1$*:* $Complaints^Q$*;* $C_2$*:* $Products^Q$*; and* $C_3$*:* $Complaints^Q \leftarrow Products^Q$*. These CNs do not include free tuple sets because of the restriction that CN cannot include free "leaf" tuple sets. (The minimum size of a CN with free tuple sets is three.) The following tuple sets are associated with our three CNs:*

$C_1$: $TS_{1,1}$

| *tupleId* | $Score(t, Q)$ |
|---|---|
| $c_3$ | *1.33* |
| $c_2$ | *0.33* |
| $c_1$ | *0.33* |

$C_2$: $TS_{2,1}$

| *tupleId* | $Score(t, Q)$ |
|---|---|
| $p_1$ | *1* |
| $p_2$ | *1* |

$C_3$: $TS_{3,1}$

| *tupleId* | $Score(t, Q)$ |
|---|---|
| $c_3$ | *1.33* |
| $c_2$ | *0.33* |
| $c_1$ | *0.33* |

$C_3$: $TS_{3,2}$

| *tupleId* | $Score(t, Q)$ |
|---|---|
| $p_1$ | *1* |
| $p_2$ | *1* |

*Following Figure 8, we first get the top tuple from each CN's tuple set and query the database for results containing these tuples (lines 5–9). Therefore, we extract (line 10) the result-tuples* $c_3$ *and* $p_1$ *from* $C_1$ *and* $C_2$ *respectively. No results are produced from* $C_3$ *since* $c_3$ *and* $p_1$ *do not join. The* $\overline{MPFS}$*s of* $C_1$*,* $C_2$*, and* $C_3$ *are 0.33, 1, and 1.17 (*$= (1.33 + 1)/2$*), respectively. Hence* $GMPFS = 1.17$*.* $c_3$ *is output since it has score* $1.33 \geq GMPFS$*. On the other hand,* $p_1$ *is not output because its score is* $1 < GMPFS$*. Next, we get a new tuple for the most promising CN, which is now* $C_3$*. The most promising tuple set for* $C_3$ *is* $TS_{3,2}$*. Therefore,* $p_2$ *is retrieved and the results of the parameterized query* $q_3(c_3, p_2)$ *(which is* $c_3 \leftarrow p_2$*) are added to* $R$*. Notice that* $q_3$ *is the query* `SELECT * FROM` $TS_{3,1}$`,` $TS_{3,2}$`, Complaints c, Products p WHERE` $TS_{3,1}.tupleId$ `=? AND` $TS_{3,2}.tupleId$ `=? AND` $TS_{3,1}.tupleId = c.tupleId$ `AND` $TS_{3,2}.tupleId = p.tupleId$ `AND` $c.prodId = p.prodId$*. Now, the* $\overline{MPFS}$ *bounds of* $C_1$*,* $C_2$*, and* $C_3$ *are 0.33, 1, and 0.67 (*$= (0.33 + 1)/2$*), respectively. Hence* $GMPFS = 1$*.* $c_3 \leftarrow p_2$ *is output because it has score*

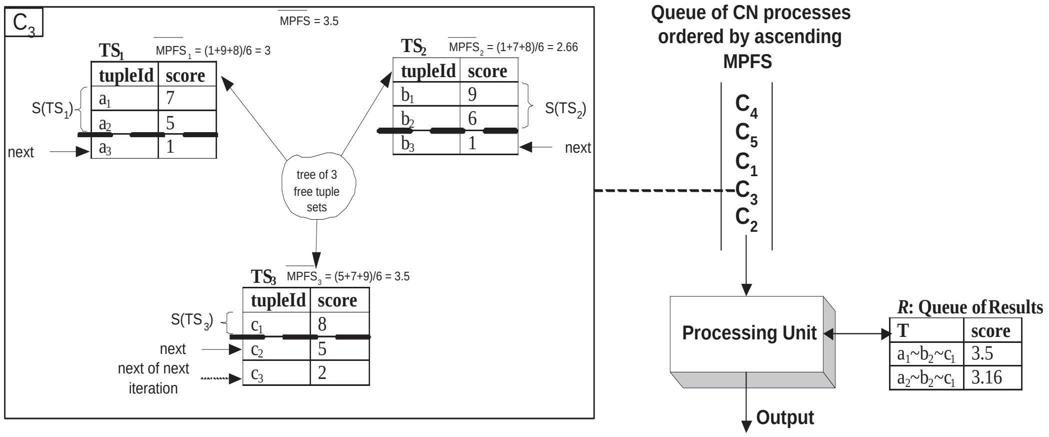
C3
$\overline{\text{MPFS}}$ = 3.5
TS1
$\overline{\text{MPFS}}_1$ = (1+9+8)/6 = 3
tupleId
score
a1
7
a2
5
a3
1
S(TS1)
next
TS2
$\overline{\text{MPFS}}_2$ = (1+7+8)/6 = 2.66
tupleId
score
b1
9
b2
6
b3
1
S(TS2)
next
tree of 3 free tuple sets
TS3
$\overline{\text{MPFS}}_3$ = (5+7+9)/6 = 3.5
tupleId
score
c1
8
c2
5
c3
2
S(TS3)
next
next of next iteration
Queue of CN processes ordered by ascending MPFS
C4
C5
C1
C3
C2
Processing Unit
Output
R: Queue of Results
T
score
a1~b2~c1
3.5
a2~b2~c1
3.16

Figure 9: Snapshot of a *Global Pipelined* execution.

$1.165 \geq GMPFS$. *Also,* $p_1$ *is output because it has score* $1 \geq GMPFS$.

Just as for *Single Pipelined*, the correctness of *Global Pipelined* relies on the combining function satisfying the tuple-monotonicity property of Definition 2. As we will see in our experimental evaluation, *Global Pipelined* is the most efficient algorithm for queries that produce many results.

### 5.5 Hybrid Algorithm

As mentioned briefly above (see Section 6.3 for more details), *Sparse* is the most efficient algorithm for queries with relatively few results, while *Global Pipelined* performs best for queries with a relatively large number of results. Hence, it is natural to propose a *Hybrid* algorithm (Figure 10) that estimates the expected number of results for a query and chooses the best algorithm to process the query accordingly.

The *Hybrid* algorithm critically relies on the accuracy of the result-size estimator. For queries with OR semantics, we can simply rely on the RDBMS's result-size estimates, which we have found to be reliable. In contrast, this estimation is more challenging for queries with AND semantics: the RDBMS that we used for our implementation, Oracle 9i, ignores the text index when producing estimates. Therefore, we can obtain from the RDBMS an estimate $S$ of the number of tuples derived from a CN (i.e., the number of tuples that match the associated join conditions), but we need to adjust this estimate so that we consider only tuple trees that contain *all* query keywords. To illustrate this simple adjustment, consider a two-keyword query $[w_1, w_2]$ with two non-free tuple sets $TS_1$ and $TS_2$. If we assume that the two keywords appear independently of each other in the tuples, we adjust the estimate $S$ by multiplying by $\frac{|TS_1^{w_1}|\cdot|TS_2^{w_2}|+|TS_1^{w_2}|\cdot|TS_2^{w_1}|}{|TS_1|\cdot|TS_2|}$, where $TS_i^w$ is the subset of $TS_i$ that contains keyword $w$. (An implicit simplifying assumption in the computation of this adjustment factor is that no two keywords appear in the same tuple.) We evaluate the performance of this estimator in Section 6.[10]

## 6 Experiments

In this section we experimentally compare the various algorithms described above. For our evaluation, we use the DBLP[11] data set, which we decomposed into relations according to the schema shown in Figure 11. $Y$ is an instance of a conference in a particular year. $PP$ is a relation that describes each paper *pid2* cited by a paper *pid1*, while $PA$ lists the authors *aid* of each paper *pid*. Notice that the two arrows from $P$ to $PP$ denote primary-to-foreign-key connections from *pid* to *pid1* and from *pid* to *pid2*. The citations of many papers are not contained in the DBLP database, so we randomly added a set of citations to each such paper, such that the average number of citations of each paper is 20. The size of the database is 56MB. We ran our experiments using the Oracle 9i RDBMS on a Xeon 2.2-GHz PC with 1 GB of RAM. We implemented all query-processing algorithms in Java, and connect to the RDBMS through JDBC. The IR index is implemented using the Oracle 9i Text extension. We created indexes on all join attributes. The same CN generator is used for all methods, so that the execution time differences reflect the performance of the execution engines associated with the various approaches. The CN generator time is included in the measured times. However, the executions times do not include the tuple set creation time, which is common to all methods.

*Global Pipelined* needs to maintain a number of JDBC cursors open at any given time. However, this number is small compared to the hundreds of open cursors that modern RDBMSs can handle. Also notice that the number of JDBC cursors required does not increase with the number of tables in the schema, since it only depends on the number of relations that contain the query keywords. In environments where cursors are a scarce resource, we can avoid maintaining open cursors by reading the whole non-free tuple sets (which are usually very small) into memory

[10] Of course, there are alternative ways to define a hybrid algorithm. (For example, we could estimate the number of results for each CN $C$ and decide whether to execute the *Single Pipelined* algorithm over $C$ or submit the SQL query of $C$ to the DBMS.) We have experimentally found some of these alternatives to have worse performance than that of the algorithm in Figure 10.

[11] http://dblp.uni-trier.de/.

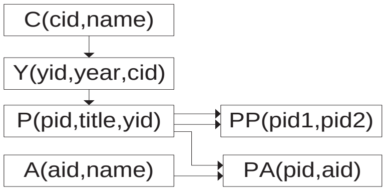

Figure 11: The DBLP schema graph ($C$ stands for "conference," $Y$ for "conference year," $P$ for "paper," and $A$ for "author").

during *Global Pipelined* execution. Furthermore, to reduce the overhead of initiating and closing JDBC connections, we maintain a "pool" of JDBC connections. The execution times reported below include this JDBC-related overhead.

The parameters that we vary in the experiments are (a) the maximum size $M$ of the CNs, (b) the number of results $k$ requested in top-$k$ queries, and (c) the number $m$ of query keywords. In all the experiments on the *Hybrid* algorithm, we set the tuning constant of Figure 10 to $c = 6$, which we have empirically found to work well. We compared the following algorithms:

- The *Naive* algorithm, as described in Section 5.1.
- The *Sparse* algorithm, as described in Section 5.2.
- The *Single Pipelined* algorithm (*SA*), as described in Section 5.3. We execute this algorithm individually for each CN, and then combine the results as in the *Naive* algorithm.
- The *Global Pipelined* algorithm (*GA*), as described in Section 5.4.
- *SASymmetric* and *GASymmetric* are modifications of SA and GA, respectively, where a new tuple is retrieved in a round robin fashion from each of the non-free tuple sets of a CN, without considering how "promising" each CN is during scheduling.
- The *Hybrid* algorithm, as described in Section 5.5.

The rest of this section is organized as follows. In Section 6.1 we consider queries with Boolean-OR semantics, where keywords are randomly chosen from the DBLP database. Then, in Section 6.2 we repeat these experiments for Boolean-AND queries, when keywords are randomly selected from a focused subset of DBLP.

### 6.1 Boolean-OR Semantics

**Effect of the maximum allowed CN size.** Figure 12 shows the average query execution time over 100 two-keyword top-10 queries, where each keyword is selected randomly from the set of keywords in the DBLP database. *GA*, *GASymmetric*, and *Hybrid* are orders of magnitude faster than the other approaches. Furthermore, *GA* and *GASymmetric* perform very close to one another (drawn almost as a single line in Figure 12) because of the limited number of non-free tuple sets involved in the executions, which is bounded by the number of query keywords. This small number of non-free tuple sets restricts the available choices to select the next tuple to process. These algorithms behave differently for queries with more than two keywords, as we show below. Also notice that *SA* and *SASymmetric* behave worse than *Naive* and *Sparse*, because the former have to evaluate the top results of every CN (even of the long ones), where the cost of the parameterized queries becomes considerable.

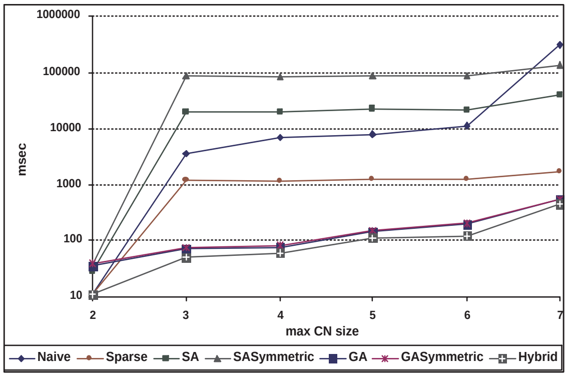

Figure 12: OR semantics: Effect of the maximum allowed CN size.

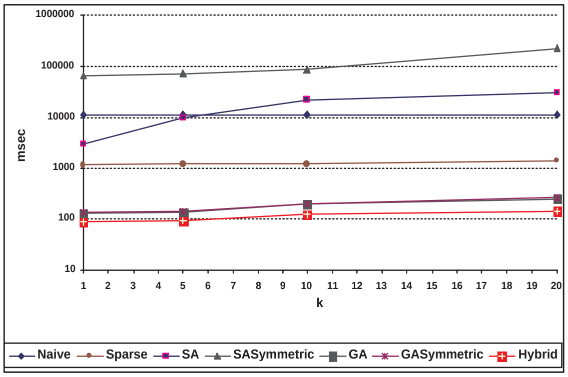

Figure 13: OR semantics: Effect of the number of objects requested, $k$.

**Effect of the number of objects requested.** Next, we fix the maximum CN size $M = 6$ and the number of keywords $m = 2$, and vary $k$. The average execution times over 100 queries are shown in Figure 13. Notice that the performance of *Naive* remains practically unchanged across different values of $k$, in contrast to the pipelined algorithms whose execution time increases smoothly with $k$. The reason is that $k$ determines the size of the prefixes of the non-free tuple sets that we need to retrieve and process. *Naive* is not affected by changes in $k$ since virtually all potential query results are calculated before the actual top-$k$ results are identified and output. The *Sparse* algorithm is also barely affected by $k$, because the values of $k$ that we use in this experiment require the evaluation of an almost identical number of CNs. Also, notice that, again, *GA* and *GASymmetric* perform almost identically.

**Effect of the number of query keywords.** In this experiment (Figure 14), we measure the performance of the various approaches as the number of query keywords increases, when $k = 10$ and $M = 6$. *SA* and *SASymmetric* are not included because they perform poorly for more than two query keywords, due to the large number of parameterized

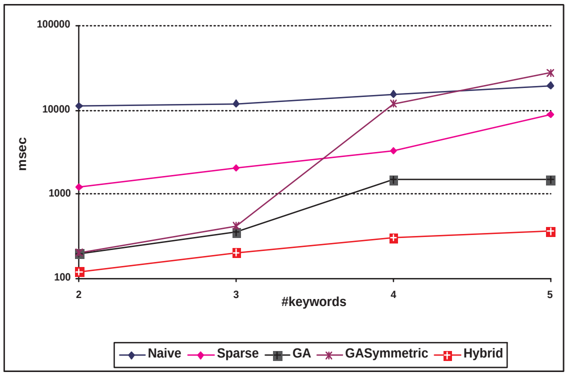

Figure 14: OR semantics: Effect of the number of query keywords.

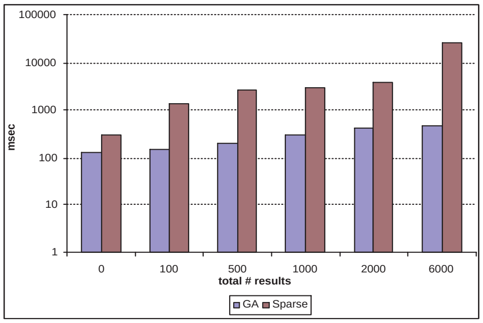

Figure 15: OR semantics: Effect of the query-result size.

queries that need to be issued. Notice that *GASymmetric* performs poorly relative to *GA*, because of the larger number of alternative non-free tuple sets to choose the next tuple from. Also notice that *Hybrid* and *GA* are again orders of magnitude faster than *Naive*. In the rest of the graphs, we then ignore *Naive*, *SA*, and *SASymmetric* because of their clearly inferior performance.

**Effect of the query-result size.** This experiment discriminates the performance of *GA* and *Sparse* by query-result size. Figure 15 shows the results of the experiments averaged over 100 two-keyword top-10 queries, when $M = 6$. The performance of *Sparse* degrades rapidly as the number of results increases. In contrast, *GA* scales well with the number of results, because it extracts the top results in a more selective manner by considering tuple trees rather than coarser CNs.

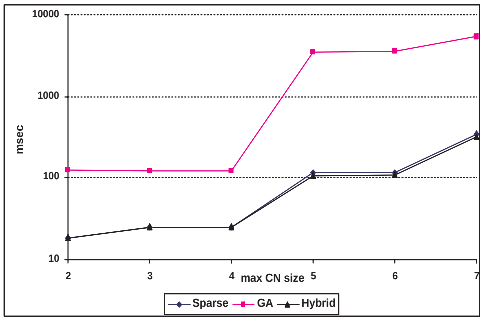

Figure 16: AND semantics: Effect of the maximum allowed CN size.

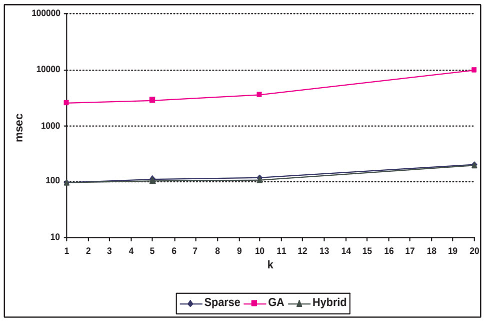

Figure 17: AND semantics: Effect of the number of objects requested, $k$.

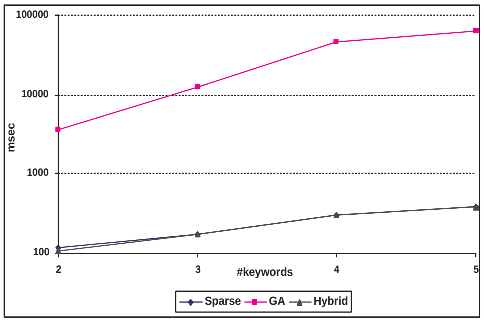

Figure 18: AND semantics: Effect of the number of query keywords.

### 6.2 Boolean-AND Semantics

We now turn to the evaluation of the algorithms for queries with Boolean-AND semantics. To have a realistic query set where the query results are not always empty, for this part of the experiments we extract the query keywords from a restricted subset of DBLP. Specifically, our keywords are names of authors affiliated with the Stanford Database Group. We compare *Sparse*, *GA* and *Hybrid*.

**Effect of $M$, $k$, and $m$.** Figures 16 ($m = 2$, $k = 10$), 17 ($m = 2$, $M = 6$), and 18 ($k = 10$, $M = 6$) show that *Hybrid* performs almost identically as *Sparse*: for AND semantics, the number of potential query results containing all the query keywords is relatively small, so *Hybrid* selects *Sparse* for almost all queries. Notice in Figure 16 that the execution time increases dramatically from $M = 4$ to $M = 5$ because of a schema-specific reason: when $M = 5$, two author keywords can be connected through the $P$ relation (Figure 11), which is not possible for $M = 4$.

**Effect of the query-result size.** Figure 19 ($m = 2$, $k = 10$, $M = 6$) shows that, unlike in Figure 15, the execution time decreases as the total number of results increases: when there are few results, the final filtering step that the algorithms perform to check that all keywords are present tends to reject many candidate results before producing the top-10 results. Figure 19 also shows that the performance of *GA* improves dramatically as the total number of results increases. In contrast, the performance of *Sparse* improves at a slower pace. The reason is that *GA* needs to process the entire CNs when there are few results for a query, which is

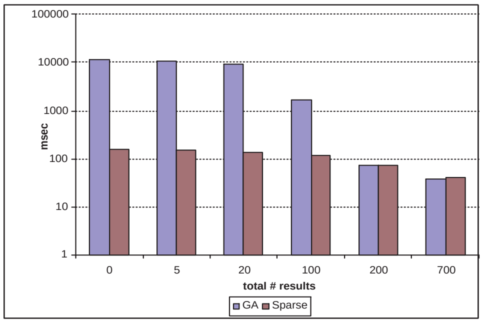

Figure 19: AND semantics: Effect of the query-result size.

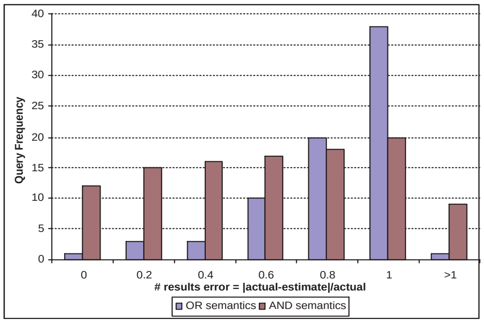

Figure 20: Quality of the result-size estimates (2-keyword queries; maximum CN size $M$=6).

more expensive than executing *Sparse* in the same setting.

### 6.3 Discussion

The main conclusion of our experiments is that the *Hybrid* algorithm *always* performs at least as well as any other competing method, provided that the result-size estimate on which the algorithm relies is accurate. (Figure 20 shows the accuracy of the estimator that we use for a set of queries created using randomly chosen keywords from DBLP.) *Hybrid* usually resorts to the *GA* algorithm for queries with OR semantics, where there are many results matching the query. The reason why *GA* is more efficient for queries with a relatively large number of results is that *GA* evaluates only a small "prefix" of the CNs to get the top-$k$ results. On the other hand, *Hybrid* usually resorts to the *Sparse* algorithm for queries with AND semantics, which usually have few results. *Sparse* is more efficient than *GA*[12] because in this case we have to necessarily evaluate virtually all the CNs. Hence *GA*, which evaluates a prefix of each CN using nested-loops joins, has inferior performance because it does not exploit the highly optimized execution plans that the underlying RDBMS can produce when a single SQL query is issued for each CN.

## 7 Conclusions

In this paper we presented a system for efficient IR-style keyword search over relational databases. A query in our model is simply a list of keywords, and does not need to specify any relation or attribute names. The answer to such a query consists of a rank of "tuple trees," which potentially include tuples from multiple relations that are combined via joins. To rank tuple trees, we introduced a ranking function that leverages and extends the ability of modern relational database systems to provide keyword search on individual text attributes and rank tuples accordingly. In particular, our ranking function appropriately combines the RDBMS-provided scores of individual attributes and tuples. As another contribution of the paper, we introduced several top-$k$ query-processing algorithms whose relative strengths depend, for example, on whether queries have Boolean-AND or OR semantics. We also presented a "hybrid" algorithm that decides at run-time the best strategy to follow for a given query, based on result-size estimates for the query. This hybrid algorithm has the best overall performance for both AND and OR query semantics, as supported by our extensive experimental evaluation over real data.

## References

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

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

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7*, 1998.

[4] N. Bruno, L. Gravano, and A. Marian. Evaluating top-$k$ queries over web-accessible databases. In *ICDE*, 2002.

[5] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In *Advances in Real Time Systems*, pages 225–248. S. H. Son, Prentice Hall, 1994.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *ACM PODS*, 2001.

[7] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. In *WWW9*, 2000.

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

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

[10] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *ACM SIGMOD*, 2001.

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

[12] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[13] A. Natsev, Y. Chang, J. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[14] M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T. Huang. Supporting ranked Boolean similarity queries in MARS. *TKDE*, 10(6):905–925, 1998.

[15] J. Plesnik. A bound for the Steiner tree problem in graphs. *Math. Slovaca*, 31:155–163, 1981.

[16] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison Wesley, 1989.

[17] A. Singhal. Modern information retrieval: a brief overview. *IEEE Data Engineering Bulletin, Special Issue on Text and Databases*, 24(4), Dec. 2001.

[12] When a query produces no results, *Sparse* has the same performance as *Naive*.