UNIVERSITY OF CALIFORNIA, SAN DIEGO

Keyword Search in Structured and Semistructured Databases

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Vagelis Hristidis

Committee in charge:

Professor Yannis Papakonstantinou, Chairperson
Professor Victor Vianu
Professor Ramamohan Paturi
Professor Dimitrios Gunopulos
Professor Dimitris Politis

2004

Copyright
Vagelis Hristidis, 2004
All rights reserved.

The dissertation of Vagelis Hristidis is approved, and it is acceptable in quality and form for publication on microfilm:

____

____

____

____

____
Chair

University of California, San Diego

2004

Dedicated to my family

and especially to my parents, Stavros and Despina.

TABLE OF CONTENTS

Signature Page . . . iii

Table of Contents . . . v

Vita, Publications, and Fields of Study . . . ix

Abstract . . . x

I Introduction . . . 1
- A. Searching Databases vs. Searching Documents . . . 1
- B. Keyword Search in Databases . . . 2
- C. Data Model . . . 4
- D. Result of a Keyword Query . . . 4
- E. Presentation of Results . . . 7
- F. Efficient Execution . . . 9
  - 1. Execution of Connection-as-Result Problem . . . 10
  - 2. Execution of Single-Object Problem . . . 12
- G. Thesis Overview . . . 13

II Definition of Result . . . 15
- A. Data Model . . . 15
- B. Problem Definition . . . 16
- C. Ranking Criteria . . . 19
- D. Ranking Factors . . . 20
  - 1. Number of Result-Trees vs. Authority Flow . . . 23
- E. Ranking Functions . . . 25
- F. Related Work . . . 28

III Efficient Execution . . . 30
- A. Architecture . . . 31
  - 1. IR Engine . . . 31
  - 2. Candidate Network Generator . . . 31
  - 3. Execution Engine . . . 32
- B. Candidate Network Generator . . . 33
  - 1. Candidate Networks Generation Algorithm . . . 37
- C. All-Results Execution . . . 41
  - 1. Greedy algorithm . . . 48
- D. Top-k Execution . . . 50
  - 1. Naive Algorithm . . . 51
  - 2. Sparse Algorithm . . . 51
  - 3. Single Pipelined Algorithm . . . 52
  - 4. Global Pipelined Algorithm . . . 56

5. Hybrid Algorithm 59
E. Experiments 60
1. CN generator 60
2. All-Results algorithms 63
3. Top-k algorithms 66
F. Related Work 73

IV Presentation of Results 76
A. Minimum Information Units 76
B. Presentation Graphs 80

V Storage of XML Data 83
A. Architecture 83
B. Decompositing XML 86
1. Decomposition Tradeoffs 88
C. Execution Stage Optimizer 93
D. Related Work 95
E. Experiments 95

VI ObjectRank 98
A. Background 98
B. Data Model and Semantics 100
1. Data Graph, Schema, and Authority Transfer Graph 100
2. Importance vs. Relevance. 103
3. Multiple-Keywords Queries. 105
4. Weigh keywords by frequency 106
5. Compare to single base set approach 107
C. Architecture 108
D. Index Creation Algorithms 110
1. General algorithm 110
2. DAG algorithm 111
3. Almost-DAG algorithm 112
4. Algorithm for graphs with small vertex cover 117
5. Serializing ObjectRank Calculation 117
6. Manipulating Initial ObjectRank values 119
E. Experiments 119
1. Quality Evaluation 119
2. Performance Experiments 121
F. Related Work 127

VII Conclusions and Future Work 131
A. Conclusions 131
B. Future Work 132

Bibliography 135

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Yannis Papakonstantinou, for his guidance and help. He has always pointed me to challenging and fresh topics, which where later pronounced as top research priorities in databases. Furthermore, his insightful comments always guided me to the right direction to solve the problem in a complete and original manner. He also brought me in contact with other top researchers, who contributed in my research progress. Finally, he gave me valuable advice on all other aspects of academic and professional life.

I would also like to thank all my coauthors for our fruitful collaboration. First, Nick Koudas who helped me in my first steps in the world of research. Michalis Petropoulos, who struggled with me to tackle a problem on which none of us was familiar. I collaborated more with Andrey Balmin whose research interests overlapped with mine. His insightful comments and development skills have been of great value to our projects. Luis Gravano has greatly contributed in our effort to formalize and solve a fuzzy and abstract problem.

Also, I would like to thank Divesh Srivastava for the time and brain power we spent together. Furthermore I would like to thank Dimitris Papadias and Yufei Tao for the project they led. Also, Tianqiu Wang whose implementation skills were invaluable for the success of the XKeyword project. Finally, I would like to thank Gautam Das, Surajit Chaudhuri and Gerhard Weikum for our fruitful collaboration during my internship at Microsoft Research.

Also, I would like to thank the database faculty of UCSD, Victor Vianu and Alin Deutsch, for keeping us motivated and making our database group one of the strongest in the country. Furthermore, the support from NSF and Eugenidou Foundation is gratefully acknowledged.

Finally, I would like to thank my family for supporting me throughout my studies.

VITA

October 13, 1976	Born, Rhodes, Greece
1999	B.S. in Electrical Engineering and Computer Science, National Technical University of Athens
2000	M.S. in Computer Science, University of California, San Diego
2004	Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

A. Balmin, V. Hristidis, Y. Papakonstantinou: Authority-Based Keyword Queries in Databases using ObjectRank. In VLDB 2004.

S. Chaudhuri, G. Das, V. Hristidis, G. Weikum: Probabilistic Ranking of Database Query Results. In VLDB 2004.

V. Hristidis, Y. Papakonstantinou: Algorithms and Applications for answering Ranked Queries using Ranked Views. VLDB Journal, Volume 13, Issue 1, January 2004

V. Hristidis, L. Gravano, Y. Papakonstantinou: Efficient IR-Style Keyword Search over Relational Databases. In VLDB 2003: pages 850-861

A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, T. Wang: A System for Keyword Search on XML Databases. In VLDB Demo Session, 2003: pages 1069-1072

V. Hristidis, Y. Papakonstantinou, A. Balmin: Keyword Proximity Search on XML Graphs. In ICDE 2003: pages 367-378

V. Hristidis, Y. Papakonstantinou: DISCOVER: Keyword Search in Relational Databases. In VLDB 2002: pages 670-681

V. Hristidis, M. Petropoulos: Semantic Caching of XML Databases. In WebDB 2002: pages 25-30

V.Hristidis, N.Koudas, Y.Papakonstantinou: PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In ACM SIGMOD 2001: pages 259-270

ABSTRACT OF THE DISSERTATION

Keyword Search in Structured and Semistructured Databases

by

Vagelis Hristidis

Doctor of Philosophy in Computer Science

University of California, San Diego, 2004

Professor Yannis Papakonstantinou, Chairperson

Keyword search on documents has been extensively studied by the Information Retrieval (IR) community. However, keyword search is becoming increasingly useful for structured and semistructured databases, due to the popularity of XML and the amount of text stored in databases. Keyword queries free the user from the requirements of knowing the database schema, the role of the keywords and a query language (SQL, XQuery).

Providing keyword search in databases is challenging on both the semantic and the performance levels. We view a database as a data graph, which captures both the relational and the XML model. A result of a keyword query is a subtree of the data graph. The factors used to rank the results are (i) the IR scores of the attribute values of the result, (ii) the structure of the result, and (iii) the authority flow between the result and the keywords through the data graph (inspired by PageRank). We show how these factors interplay and how they can be combined in meaningful ways that allow efficient execution methods.

On the performance level, we present efficient algorithms to produce all or the top-k results of a keyword query. We study two models: the middleware model where the system lies on top of an already operational database system to provide keyword querying, and the dedicated system where we handle the storage of the data and precompute various data to offer real-time response times. The execution techniques are thoroughly experimentally evaluated. Finally, we present a novel technique to present the results to the user.

Chapter I

Introduction

I.A Searching Databases vs. Searching Documents

Databases and Information Retrieval (IR) have followed distinct ways, mainly due the fact that databases used to store only rigidly structured data and handle rigidly structured queries, which serve the purposes of particular well-designed applications. In contrast, IR is employed for information discovery and primarily studies how (unstructured) documents are ranked according to their relevance to an unstructured query (typically a set of keywords).

However, due to the increasing availability of database data, the increasing popularity of XML and the increasing amount of text stored in databases, it has become imperative to allow unstructured queries on structured and semistructured databases, in addition to documents. The most popular type of unstructured query is keyword queries, whose success is proven by Web search engines.

Keyword search is the most popular information discovery method because the user does not need to know either a query language or the underlying structure of the data. The search engines available today provide keyword search on top of sets of documents. In addition to documents, a huge amount of information is stored in databases, but no simple way is available to discover information in them, except by using structured languages (such as XQuery or SQL) where

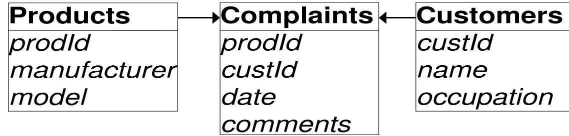


Figure I.1: Schema of the *Complaints* database.

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

the type(s) of each keyword must be specified, as well as the types of connections between the objects.

I.B Keyword Search in Databases

Consider a customer-service database from a large vendor of computer equipment. Figure I.1 shows the schema of the database, while Figure I.2 shows a possible instance of the database. For example, the table *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*). The first tuple of this table corresponds to a complaint by customer *c3232* about product *p121*, which, 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. Furthermore, even for the case when the desired results consist of a single tuple, this approach has the drawback that the link-structured of the graph is ignored (see Chapter VI).

In the above example, 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

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

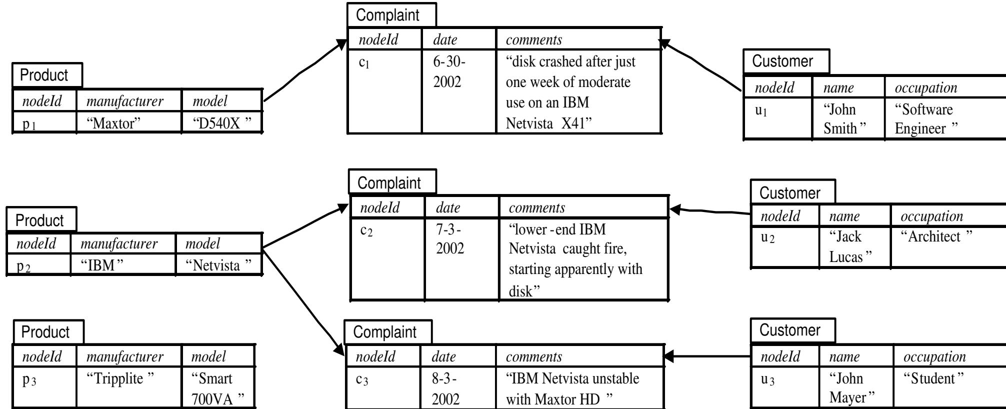


Figure I.3: Data graph of the *Complaints* database.

the *Complaints* relation).

I.C Data Model

We view a database as a labeled graph, which is a model that easily captures both relational and XML databases. Every node (object) v has a set of attributes, which represent the relational or XML attributes for relational and XML databases respectively. A node of the *data graph* corresponds to a tuple or XML node for relational and XML databases respectively. The edges typically correspond to primary to foreign key relationships in relational databases, although other types of semantic connections could be used as well. In XML databases, the edges correspond to containment or ID-IDREF edges. Figure I.3 shows the data graph corresponding to the database instance of Figure I.2. The data graph conforms to a schema graph (Figure I.1) as we explain in Section II.A.

I.D Result of a Keyword Query

Free-form keyword search on databases has attracted recent research interest. DBXplorer [6] works on relational databases, and given a keyword query,

it joins 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. BANKS [9] assumes that the database is a graph with no schema and calculates subtrees that contain the keywords. On the other hand, Goldman et al. [22] define the result of a keyword query to be a set of nodes, which are closest to the set of nodes that contain the keywords.

In this text we define two different versions of the problem to cover both approach classes. The first version is the *connection-as-result* problem, where we return a list of subtrees (result-trees) of the data graph, and the second is the *single-object* version, where the answer is a set of nodes of the data graph ranked according to their relevance to the query. The reasons why we treat the single-object as a separate case are the following. First, queries with single objects (nodes) as results are more intuitive and common. For example, Web search engines return single pages instead of trees of interconnected pages. Second, given that attribute values typically contain little text, it is unlikely that all the keywords of a query will be found in a single node. On the other hand, it is likely that all keywords are contained in a result-tree with an adequately big size. Hence, to avoid getting empty results often, a different approach is required.

To rank the results, we use three factors, which to the best of our knowledge include all the ranking methods proposed in prior work. The first factor is the IR score of the individual attribute values of the nodes[2]. We chose the attribute granularity (as opposed to tuple granularity) because the attribute value is the smallest information unit in a database and usually does not depend on the schema design decisions. To calculate the attribute value IR scores we leverage state-of-the-art IR relevance-ranking functionality already present in modern RDBMSs, which allows the generation of 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 of Figure I.2 with a high relevance

[2] In Chapter VI we assume for simplicity that every node contains only one attribute.

score, after word stemming (so that "crash" matches "crashed") and stop-word elimination (so that the absence of "a" is not weighed too highly).

The second factor is the structure of the result-tree. For example, we could define the score of a tree to be inversely proportional to its size, since smaller trees usually denote tighter semantic connection[3]. Another approach is to allow an expert to rank the schemata of the results according to their semantic values. For example, in an enterprise database, connecting a customer and a supplier through an order denotes a tighter connection than connecting them through a common nation where they do business.

The third factor is the authority flow between the nodes of the data graph. This influences the degree of association (i) among the nodes of the result-tree, and (ii) between the nodes of the result-tree and the nodes with the keywords. The authority flow represents the probability that starting from a node of the data graph we will reach another node, by following the edges of the data graph (see Section VI.A for more details). This idea in inspired by PageRank [11], which is used by Google[4] to assign a global importance to the pages of the Web. For example, if many customers with occupation "architect" issue a complaint about product "Netvista", there is a high association between the keyword "architect" and the node "Netvista".

All factors can be applied to both versions of the problem. However, in this text, for the connection-as-result problem we focus on ranking methods that combine the IR scores of the individual attribute values and the structure of the result-tree (Chapter III), which are the factors adopted by relevant work [6, 9] as well. For the single-object version, the structure of the result-tree is trivial. Hence, to produce a high-quality ranking of the result nodes we focus on the degree of association between the result nodes and the nodes that contain the keywords, applying the authority flow factor (Chapter VI).

---

[3] Notice that this factor is not applicable to the single-object version of the problem except if we assign different importance to different types of nodes.

[4] http://www.google.com

We have created online demonstrations for the connection-as-result and single-object problems on the DBLP[5] publications database available at http://www.db.ucsd.edu/XKeyword and http://www.db.ucsd.edu/ObjectRank respectively.

I.E Presentation of Results

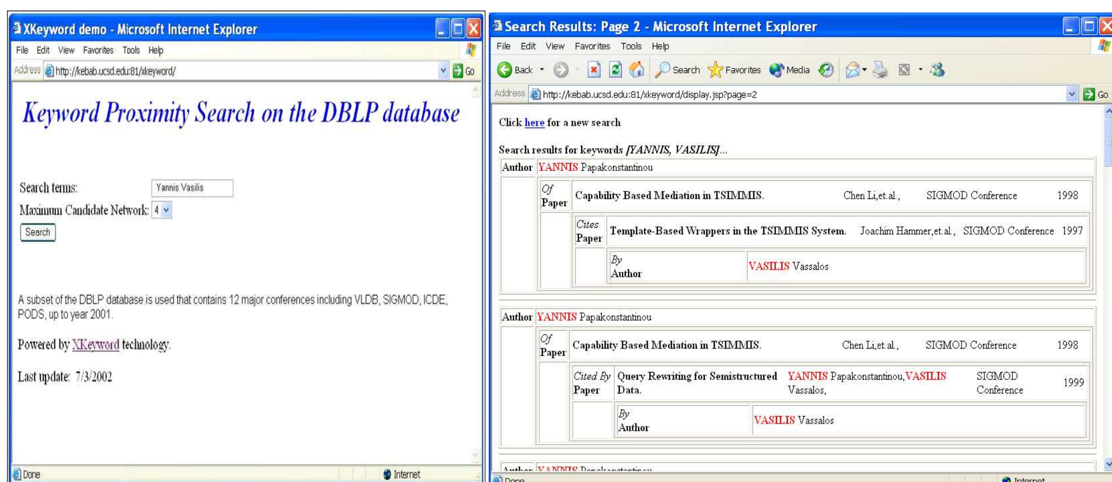
The presentation of the results faces two key challenges that have not been addressed by prior systems. First, the results need to be semantically meaningful to the user. Towards this direction, XKeyword [33] associates a minimal piece of information, called *target object*, to each node and displays the target objects instead of the nodes in the results. In the DBLP demo (Figure I.4) available at http://www.db.ucsd.edu/XKeyword, XKeyword displays target object fields such as the paper title and conference along with a paper. In the Complaints database example we can view every tuple as a target object.

The second challenge is to avoid overwhelming the user with a huge number of often trivial results, as is the case of DBXplorer [6], which presents all trees that connect the keywords. In doing so, they produce a large number of trees that contain the same pieces of information many times. For example, consider the keyword query "Netvista Smith" and the instance of the Complaints database shown in Figure I.5, where we only show the type and the keywords of each node. This instance contains four results:

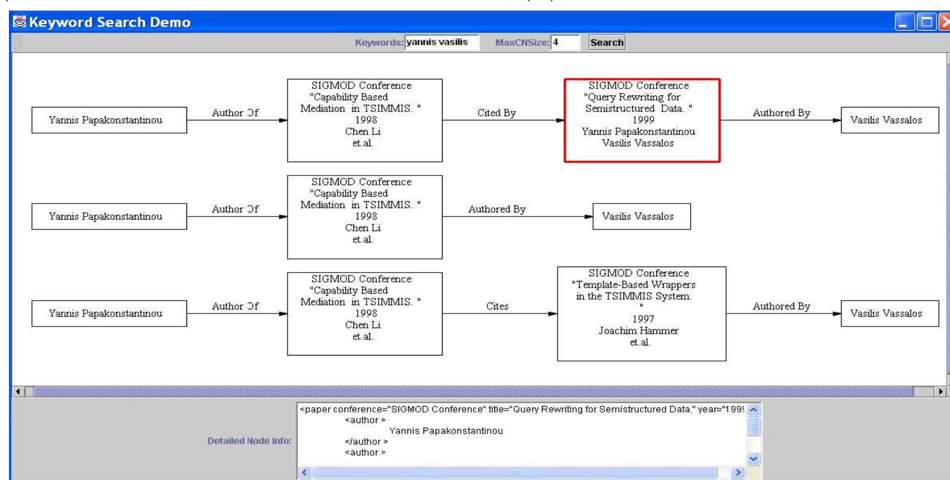
$$\begin{aligned} &N_1 : c_1 \leftarrow p_1 \rightarrow c_3 \rightarrow u_1, \quad N_2 : c_1 \leftarrow p_2 \rightarrow c_2 \rightarrow u_2, \\ &N_3 : c_1 \leftarrow p_2 \rightarrow c_2 \rightarrow u_1, \quad N_4 : c_1 \leftarrow p_1 \rightarrow c_2 \rightarrow u_2 \end{aligned} \tag{I.1}$$

The above results contain a form of redundancy similar to multivalued dependencies [54]: we can infer N_3 and N_4 from N_1 and N_2. In that sense, N_3 and N_4 are trivial, once N_1 and N_2 are given. Such trivial results penalize performance and overwhelm the user.

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



(a) Query page (b) Presentation as a list of results



(c) Presentation using Presentation graphs

Figure I.4: XKeyword demo

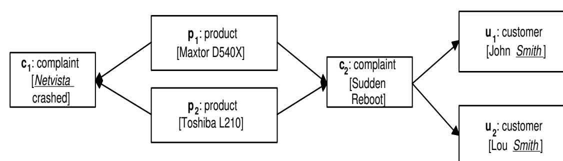


Figure I.5: Multivalued dependencies in results

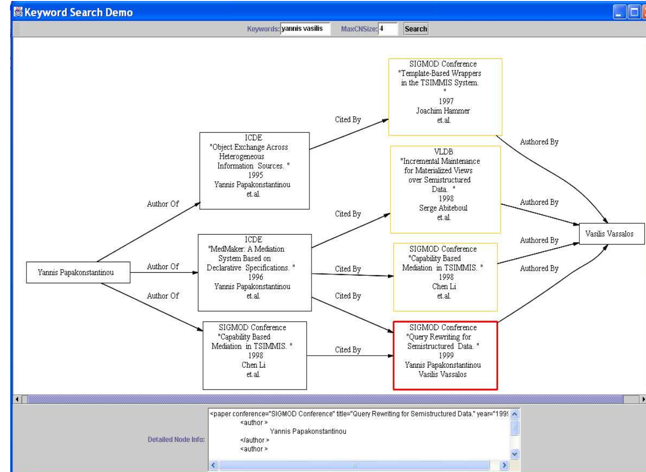


Figure I.6: Presentation graph

XKeyword uses a novel presentation method, where the results are grouped by their structure (schema). For every result schema s, a *presentation graph* is generated which contains all results of type s. At any point only a subset of the graph is shown (see Figure I.4 (c)), as it is formulated by various navigation actions of the user. Initially the user sees one result-tree r_0. By clicking on a node of interest the graph is expanded to display more nodes of the same type that belong to result-trees that contain as many as possible of the other nodes of r_0. Towards this purpose we define a minimal expansion concept. For example, clicking on the outlined paper node of Figure I.4 (c) displays all nodes of papers written by "Vasilis" and for which there is a paper of "Yannis" cited by them, as shown in Figure I.6.

I.F Efficient Execution

As mentioned in Section I.D, this text tackles two problem instances: the connection-as-result problem where the IR scores of the attribute values along with the tree structured are used to rank the results, and the single-object problem where the random walk (authority flow) principles are used to rank the results.

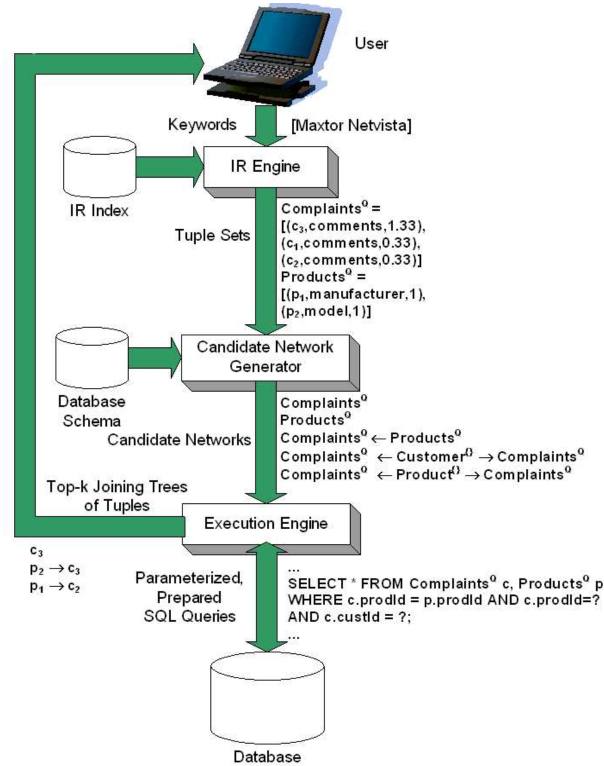


Figure I.7: Architecture of connection-as-result system [30].

Hence we study methods to optimize the execution of these problems.

I.F.1 Execution of Connection-as-Result Problem

The work on the performance level presented in this work is divided into two classes. The first class [32, 30] consists of execution algorithms which operate at a middleware level. That is, they operate on top of an already functioning database system, and their only requirements are the existence of a full-text index on top of the database and a querying interface to the database system. The second class [33] of work focuses on how to store the data in order to allow efficient keyword search. In particular, we describe methods to store XML data into relations, although the same ideas can be applied to other data models as well.

Execution Algorithms As we see in Figure I.7 (described in more detail in Section III.A), the execution consists of three steps. First, the IR index of the DBMS is queried to retrieve the locations of the keywords in the database. Second, the Candidate Network Generator creates all possible ways to connect the keywords using the keywords' locations and the database schema. Each Candidate Network (CN) corresponds to a join expression query. We present a CN Generator algorithm which is complete (every result is produced by a CN) and non-redundant (if we remove a CN, there is a database instance for which we miss a result). Finally, the CNs (queries) are input to the Execution Engine which outputs a sequence of queries to the database.

The Execution Engine module is the most challenging as we have found that different methods are appropriate for different requirements of the query. In particular, if we want to produce *all* the results (of size up to M), we have found that the most efficient method is to submit the queries corresponding to the CNs directly to the DBMS. Due to the nature of the problem, the CNs share join expressions. This offers an opportunity to build a set of intermediate results and use them in the computation of multiple candidate networks. We show that the selection of the optimal sequence of intermediate results to build is NP-complete on the sum of the sizes of the CNs. Hence, we present a greedy algorithm that produces an execution plan that calculates and uses intermediate results in evaluating the CNs, which we show that performs very well. Finally an SQL statement is produced for each line of the execution plan and these statements are passed to the DBMS.

On the other hand, as is common in most IR applications, the user may be interested only in the top-k results, since the number of results may be too big as we discussed in Section I.E. We present and experimentally evaluate algorithms that exploit this user behavior and efficiently produce the top ranked results for a wide class of ranking functions.

Data Storage The second class of optimization techniques assumes we have control on the way the data is stored. This is particularly true in the case of XML data, since there is not yet a standard way to store them. In this work we study how XML data should be stored in a relational database system [10, 50, 20, 40, 16, 48, 8] to enable an efficient computation of the connections between the keywords. XKeyword builds a set of *connection relations*, which precompute particular path and tree connections on the schema graph. Connection relations are similar to path indices [18] since they facilitate fast traversal of the database, but also different because they can connect more than two objects and they store the actual path between a set of target objects, which is needed in the answer of the keyword query. A core problem is the choice of the set of connection relations that are precomputed.

During execution, the optimizer of the Execution Engine inputs the CNs and generates an *execution plan*. The key challenges of the optimizer are (a) to decide which connection relations to use to efficiently evaluate each CN and (b) to exploit the reusability opportunities of common subexpressions among the CN's (the latter is described above). Both decisions, which are NP-complete, dramatically affect the performance as we show experimentally.

I.F.2 Execution of Single-Object Problem

Calculating the ObjectRank values in runtime is a computationally intensive operation, especially given the fact that multiple users query the system. This is resolved by precomputing an inverted index where for each keyword we have a sorted list of the nodes with non-trivial ObjectRank for this keyword. During run-time we employ the *Threshold Algorithm* [17] to efficiently combine the lists. However, our approach induces the cost of precomputing and storing the inverted index. Regarding the space requirements, notice that the number of keywords of a database is typically limited, and much less than the number of users in a personalized search system [36]. Furthermore, we do not store nodes with ObjectRank

below a threshold value (chosen by the system administrator), which offers a space versus precision tradeoff. In Section VI.E.2 we show that the index size is small relative to the database size for two bibliographic databases.

Regarding the index computation, we present and experimentally evaluate two classes of optimizations. First, we exploit the structural properties of the database graph. For example, if we know that the objects of a subgraph of the schema form a Directed Acyclic Graph (DAG), then given a topological sort of the DAG, there is an efficient straightforward one-pass ObjectRank evaluation. We extend the DAG case by providing an algorithm that exploits the efficient evaluation for DAGs in the case where a graph is "almost" a DAG in the sense that it contains a large DAG subgraph. In particular, given a graph G with n nodes, which is reduced to a DAG by removing a small subset of m nodes, we present an algorithm which reduces the authority calculation into a system of m equations - as opposed to the usual system of n equations. Furthermore, we present optimization techniques when the data graph has a small vertex cover, or if it can be split into a set of subgraphs and the connections between these subgraphs form a DAG.

Second, notice that the naive approach would be to calculate each keyword-specific ObjectRank separately. We have found that it is substantially more efficient to first calculate the global ObjectRank, and use these scores as initial values for the keyword-specific computations. This accelerates convergence, since in general, objects with high global ObjectRank, also have high keyword-specific ObjectRanks. Furthermore, we show how storing a prefix of the inverted lists allows the faster calculation of the ObjectRanks of all nodes.

I.G Thesis Overview

Chapter II presents the data model and formalizes the problem. Chapter III presents algorithms to answer keyword queries when the system is a middleware on top of an already operational database. Then, Chapter IV describes

methods to present the results to a user in a meaningful way. Chapter V shows how we can store XML data in a relational database system to allow efficient keyword querying. Then, Chapter VI discusses in detail the authority flow factor. Finally, we conclude and present future work in Chapter VII.

Chapter II

Definition of Result

This chapter presents the data model and defines the solution to a keyword query. In particular, Section II.A describes our data model. Then, Section II.B formally defines the problem. Sections II.C and II.D describe the user criteria used to rank the results and the factors to implement these criteria on the database. Finally Section II.F presents related work.

II.A Data Model

We view a database as a labeled graph, which is a model that easily captures both relational and XML databases. The *data graph* $D(V_D, E_D)$ is a labeled directed graph where every node (referred as tuple in Chapter III where we focus on relational data, and object in Chapter VI) v has a label $\lambda(v)$, a *nodeid* and a set of attributes. These attributes correspond to relational or XML attributes for relational or XML databases respectively. Notice that we omit the primary and foreign key attributes on the data graph, since their meaning is captured by the edges. For example in the data graph of Figure I.3, node p_1 has label *Product* and attributes *manufacturer* and *model*. Each attribute contains a list of keywords. For example, attribute *occupation* of node u_1 contains the keyword list "Software, Engineer". In Chapter VI, we simplify by merging all attribute keyword lists of each node into a single set of keyword, since we only rank according to the link-

structure of the data graph.

One may assume richer semantics by including the metadata of a node in the list of keywords. For example, the metadata "Product", "manufacturer", "model" could be included in the keywords of node p_1. The specifics of modeling the metadata of a node are orthogonal to this work and will be neglected in the rest of the text.

Each edge e from u to v is labeled with its *role* $\lambda(e)$ (we overload λ) and represents a relationship between u and v. For example, every "Product" to "Complaint" edge of Figure I.3 has the label "has complaint". When the role is evident and uniquely defined from the labels of u and v, we omit the edge label. For simplicity we will assume that there are no parallel edges and we will often denote an edge e from u to v as "$u \rightarrow v$".

The *schema graph* $G(V_G, E_G)$ (Figure I.1) is a directed graph that describes the structure of the data graph D. Every node has an associated label. Each edge is labeled with a role, which may be omitted, as discussed above for data graph edge labels. We say that a data graph $D(V_D, E_D)$ *conforms* to a schema graph $G(V_G, E_G)$ if there is a unique assignment μ such that:

1. for every node $v \in V_D$ there is a node $\mu(v) \in V_G$ such that $\lambda(v) = \lambda(\mu(v))$;
2. for every edge $e \in E_D$ from node u to node v there is an edge $\mu(e) \in E_G$ that goes from $\mu(u)$ to $\mu(v)$ and $\lambda(e) = \lambda(\mu(e))$.

For relational databases, where we store the data for our prototypes and experiments, each node s of the schema graph corresponds to a relation R and each edge to a primary to foreign key relationship.

II.B Problem Definition

Given a data graph D, we define a *node-tree* T as a subtree of D. The *size* of a node-tree is the number of nodes it contains. For relational databases, a node-tree corresponds to a *joining tree of tuples*, which is defined as follows.

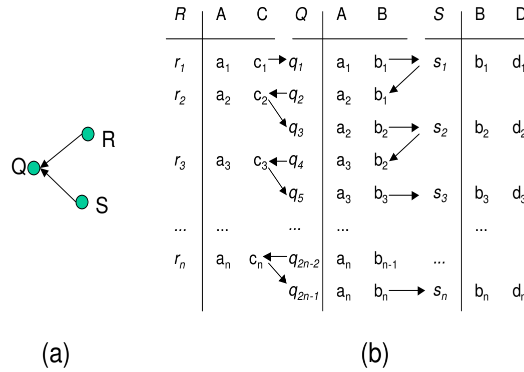


Figure II.1: A many-to-many relationship

Definition 1 (Joining tree 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) $t_i \bowtie t_j \in R_i \bowtie R_j$. Also, no tuple t is contained more than once in T. The size(T) of a joining tree T is the number of tuples in T.*

A *keyword query* is a list of keywords $Q = [w_1, \ldots, w_m]$. The result for such a query is a list of node-trees T (which we call *result-trees*) ordered by their $Score(T, Q)$ score for the query, where $Score(T, Q)$ is discussed below. (Ties are broken arbitrarily.) We require that any node-tree T in a query result be minimal: if a node t with zero score is removed from T, then the nodes remaining in T are "disconnected" and do not form a tree. In other words, T cannot have a leaf node with zero score.

A top-k keyword query is a keyword query where only the first k results are returned to the user. As an example, for a choice of ranking function *Score* the results for a top-3 query [Netvista Maxtor] over our *Complaints* database of Figure I.3 could be (1) c_3; (2) $p_2 \rightarrow c_3$; and (3) $p_1 \rightarrow c_1$. Finally, we do not allow any node to appear more than once in a node-tree.

If there is a many-to-many relationship between two nodes (relations for relational databases) of the schema graph, then the result-trees could have an arbitrarily big size, which is only data bound. Figure II.1 shows an extreme case

where there is a many-to-many relationship between two relations R, S. There is a foreign to primary key relationship from Q to R and from Q to S on the homonymous attributes. Suppose that attribute values c_1 and d_n contain the two keywords of a query. The joining tree of tuples $r_1 \bowtie q_1 \bowtie s_1 \bowtie q_2 \bowtie r_2 \bowtie \cdots \bowtie r_k \bowtie q_{2k-1} \bowtie s_k \bowtie q_{2k} \bowtie \cdots \bowtie r_n \bowtie q_{2n-1} \bowtie s_n$ uses all tuples from all three relations, as shown by the arrows in Figure II.1. So we see that the size of the joining tree of tuples can only be bound by the dataset when there are many-to-many relationships. Hence, we typically limit the maximum size M of a result-tree to bound the total number of results.

An interesting special case arises for $M = 1$, that is, when the results are single nodes. This is defined as a separate sub-problem for two reasons: First, queries with single objects (nodes) as results are more intuitive and common. For example, Web search engines return single pages instead of trees of interconnected pages. Second, given that attribute values typically contain little text, it is unlikely that all the keywords of a query will be found in a single node. On the other hand, it is likely that all keywords are contained in a result-tree with an adequately big size. Hence, to avoid getting empty results often, a different approach is required when $M = 1$. We call this version of the problem *single-object* keyword search, which is discussed in more detail in Chapter VI.

The problem version for $M > 1$ is called *connection-as-result* keyword search, because the actual connections determine the score of a result. The key challenge of this version is how to find the best possible ways to associate nodes that contain subsets of the keywords of the query, in order to get a result-tree that constitutes a meaningful answer to the query. This problem is the focus of Chapters III-V.

Notice that this text focuses on databases that conform to a schema. However, most of the ranking discussions of this chapter apply to semistructured and unstructured databases as well.

II.C Ranking Criteria

The ranking of a result-tree is subject to the criteria of every user. For example in a complaints database (Figure I.1), a user may want to rank product-complaint connections according to how related the nodes are to the keywords of the query, or according to the global importance of the products in the database and so on. We believe that the following criteria are close to the criteria that a typical user would have in mind:

Global Importance. Every node in a database can be assigned a global importance for a specific application. For example in a complaints database, we could assign to every customer an importance proportional to their amount of spending. On the other hand, in a publications database, the global importance of every paper is the amount of authority flowing from other papers through citation edges (see Chapter VI), similarly to the calculation of PageRank [11] for Web pages.

Relevance to Keywords. Every node u can be relevant to the keywords of the query in two ways. First, u may contain some of the keywords, or more generally be relevant to the keywords according to an Information Retrieval (IR) module, which may consider stemming, synonyms and so on. This way corresponds to the IR attribute value scores described below. Second, u may be connected to nodes relevant to the keywords of the query. This relevance may be captured by the probability that a random surfer of the data graph starting with the nodes relevant to the keywords will reach u (Chapter VI).

Association between Nodes of Result-Tree. The score of a result-tree T is higher if there is a tight semantic association between its nodes. The tightness of this association can be defined either using only the edges of T or using the whole data graph. The first approach [32, 6, 33, 30] is more appropriate for the connection-as-result problem, because the actual structure of the result-tree is

ranked. For example in a publications database, suppose that for the keyword query $[w_1, w_2]$ there are three result-trees $paper_1^{w_1} \rightarrow paper_2 \rightarrow paper_3^{w_2}$, $paper_1^{w_1} \rightarrow paper_4 \rightarrow paper_5^{w_2}$, $paper_1^{w_1} \rightarrow paper_6 \rightarrow paper_5^{w_2}$ where the superscript w denotes the containment of keyword w. Then, according to the first approach, all result-trees receive the same score since they have the same structure. However, according to the second approach the last two result-trees receive a higher score because there is a tighter connection between $paper_1$ and $paper_4$ (through $paper_4, paper_6$) than between $paper_1$ and $paper_3$ (only through $paper_3$). A combination of the two approaches is also possible.

Specificity. Typically a user prefers results that are specifically relevant to the keywords as opposed to results that are relevant to a wide range of topics. The specificity criterion can be included in the IR score of the attribute values explained below. Furthermore, in XRANK [26] the specificity is measured using a decay factor proportional to the distance in terms of containing subelements of an XML element from a keyword. The specificity criterion can staightforwardly be integrated in our framework and is ignored in the rest of the text for simplicity and space constraint reasons.

II.D Ranking Factors

Section II.C presented the criteria that a user can use to customize his/her ranking. However, these criteria are abstract and cannot be straightforwardly applied to the data graph to answer a keyword query. This section presents more primitive ranking factors which can be combined to implement the ranking criteria of Section II.C. To the best of our knowledge, these factors include all the ranking methods proposed in prior work [22, 9, 6, 26]. The ranking factors are the following.

IR score of attribute values. The first factor is the IR score of the individual attribute values of the nodes[1]. We chose the attribute granularity (as opposed to tuple granularity) because the attribute value is the smallest information unit in a database and usually does not depend on the schema design decisions. To calculate the attribute value IR scores we leverage state-of-the-art IR relevance-ranking functionality already present in modern RDBMSs, which allows the generation of 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 of Figure I.2 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).

The functions used are traditional IR functions [47] which use factors like term frequency (tf), inverse document frequency (idf), document length (dl) and others. As an example, a state-of-the-art IR definition for a single-attribute scoring function *Score* is as follows [51]:

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

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).

Structure of Result-Tree. The second factor is the structure (that is, schema) of the result-tree. One intuitive approach for many applications would be to define the score of a tree to be inversely proportional to its size, since smaller trees usually denote tighter semantic connection[2]. Another approach is to allow an expert to rank the schemata of the results according to their semantic values. For example, in an enterprise database, connecting a customer and a supplier through

[1] In Chapter VI we assume for simplicity that every node contains only one attribute.

[2] Notice that this factor is not applicable to the single-object problem except if we assign different scores to different types of nodes.

an order node denotes a tighter semantic connection than connecting them through a common nation where they do business.

Authority Flow. The third factor reflects the influence of the link-structure of the data graph in the ranking of the results. A typical mathematical notion to measure the flow of authority between the nodes of the data graph is the random walk probability of reaching a node starting from another node. This factor is inspired by PageRank [11] and can be used to evaluate any of the criteria of Section II.C. For example, to evaluate the global importance of a paper u in a publications database, we calculate the probability that starting from any node of the graph we are at u at a given time following a random outgoing edge in every step. Similarly, to measure the relevance of u to a keywords w contained in node v, we calculate the probability that starting from v we are at u at a given time. For example, the "Data Cube" paper [24] is cited by many papers relevant to "OLAP", so it receives a top relevance ranking for the keyword query [OLAP]. Notice that the authority flow factor is only applicable to databases where there is a natural flow of authority between the nodes (e.g.: complaints database, publications database, the Web and so on).

A large number of problems can be defined by considering subsets of the ranking factors for the single-object and the connection-as-result problems. In this text we thoroughly study two such problems, which we believe are intuitive and provide the base to tackle other problems as well. In Chapters III-V we use the first two ranking factors (IR scores of the attribute values and structure of result-tree) to answer the connection-as-result problem (a subset of these factors was used in relevant work [6, 9] as well). In Chapter VI we use the authority flow factor to answer single-object queries, since the structure of the result-tree factor does not make much sense for single-object results.

The attribute values' IR score factor is applicable to both the single-object and the connection-as-result problems, and is orthogonal to the other two

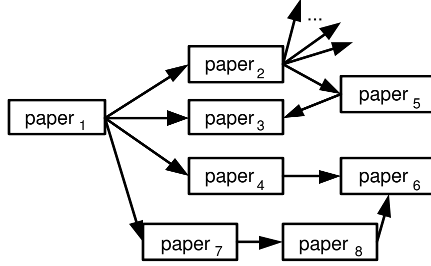


Figure II.2: Example showing the relation between number of result-trees and authority flow

factors. On the other hand, there is an interplay between the structure of the result-tree and the authority flow factors as we discuss in Section II.D.1.

II.D.1 Number of Result-Trees vs. Authority Flow

In previous work, the authority flow factor [11, 29, 45] and the proximity (structure of result tree) factor [22, 9, 6] were considered fundamentally different ways of ranking the results of a keyword query. However, as we explain in this section, there is interplay between them. More specifically, there is a correlation between the authority flow between two nodes and the number of result-trees containing them: When there are many result-trees connecting nodes u, v then there is a good chance that there is high flow of authority between u, v, and inversely. Hence, we can approximate the authority flow between u, v by counting (in a weighted manner) the result-trees that contain them.

In order to use this approximation we must take into consideration the probabilities of moving between adjacent nodes of the result-trees. For example in Figure II.2, the result-tree $T_1 : \mathit{paper}_1 \rightarrow \mathit{paper}_2 \rightarrow \mathit{paper}_5$ represents less authority transfer from paper_1 to paper_5 than $T_3 : \mathit{paper}_1 \rightarrow \mathit{paper}_4 \rightarrow \mathit{paper}_6$ does from paper_1 to paper_6. Also, $T_2 : \mathit{paper}_1 \rightarrow \mathit{paper}_3 \rightarrow \mathit{paper}_5$ carries no authority from paper_1 to paper_5, because authority only flows along the direction of the edges (authority only flows to cited papers and not to citing as we explain in Chapter VI).

However, this approximation suffers from the *size error* E_S, which is due to the fact that we limit the size of the result-trees to M, but authority may flow along longer paths. In the example above, the path T_4 : $paper_1 \rightarrow paper_7 \rightarrow paper_8 \rightarrow paper_6$ is ignored since it has size greater than $M = 3$.

Estimation of E_S As we explain in Chapter VI, the authority flow is calculated using the power method (Equation VI.1), which is an iterative method which calculates the primary eigenvalue of a matrix $\mathbf{A}$. To estimate E_S we make the following observation. Taking into account the result-trees of size up to M is equivalent to executing the power method for M iterations. Hence, E_S is equal to the error of stopping the power method after M iterations. This problem has been studied by the numerical linear algebra community [39], where they prove that for positive symmetric (in the general case $\mathbf{A}$ is not symmetric, but no result is available for asymmetric matrices) matrices the average relative error is $E_S = O(ln(n)/k)$, where n is the dimension of $\mathbf{A}$ (number of nodes of the data graph, $n = |V_D|$) and k is the number of iterations (we set $k = M$ for the estimation of E_S).

Error in ordering E_S is the relative error of the authority of a specific node v with respect to a base set consisting of the node u. However, since we are solving a ranking problem, the ordering of the nodes with respect to a base set is more important and not the actual values of the authorities of the nodes. That is, we want to calculate the probability $P(v, v') = P(r(v) > r(v')|a(v) > a(v'))$ that a node v has a higher actual authority $r(.)$ than v', given that v has a higher approximate authority $a(.)$.

To simplify the calculation of $P(v, v')$ we assume that (i) the absolute error E_A for both v and v' is the same $E_A = E_S \cdot (a(v)+a(v'))/2$ ($\cong E_S \cdot a(v) \cong E_S \cdot a(v')$), and (ii) the value of $r(v)$ ($r(v')$) is uniformly distributed between $a(v) - E_A$ and $a(v) + E_A$ ($a(v') - E_A$ and $a(v') + E_A$). Using these assumptions we can prove

that

$$P(v,v') = 0.5 + \frac{(a(v)-a(v'))^2}{8 \cdot E_A^2} = 0.5 + 0.5 \cdot \left(\frac{a(v)-a(v')}{a(v)+a(v')}\right)^2 \cdot \frac{1}{E_S^2} \qquad \text{(II.2)}$$

Using the above value for E_S, we have

$$P(v,v') = 0.5 + 0.5 \cdot \left(\frac{a(v)-a(v')}{a(v)+a(v')}\right)^2 \cdot \frac{1}{(O(ln(n)/M))^2} \qquad \text{(II.3)}$$

From this equation it is clear that as the maximum size M of result-trees increases, $P(v,v')$ goes to 1. Notice that the above analysis also applies for base sets consisting of more than one nodes, since the error bound E_S does not generally depend on the size of the base set.

II.E Ranking Functions

In Section II.C we discussed what criteria can be used to customize the ranking scheme to a specific application. Then, in Section II.D we present how these criteria can be implemented on the data graph. However, so far we have not presented any concrete way to combine these ranking factors to bring an order to the results of a query. This section discusses how these factors can be combined into meaningful ranking functions, which also allow efficient execution methods as we show in Chapter III. In some parts of this section we switch from general graph databases to relational databases, since they are used to store the data of our prototypes.

The class of ranking functions described combine all criteria of Section II.C. However, regarding the criterion of association between the nodes of the result-trees, we only consider the structure of the result-tree and not the association between the nodes of the result-tree through the whole data graph. The reason we have ignored this sub-criterion in this thesis is that computing all the pairwise association degrees between all nodes of the data graph is too expensive to precompute and store. Furthermore, in this case the semantics become unclear for more than two keywords.

Before stating the ranking function for the connection-as-result problem, which is the focus of this section, we present how previous works have addressed the problem of ranking the results of a keyword query. Given a query Q, both DISCOVER [32] and DBXplorer [6] assign a score to a result-tree 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 [9] uses the following scoring scheme:[3]

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

where $f_r(T)$ measures how "related" the types (relations in relational databases) of the nodes (tuples) of T are, $f_n(T)$ depends on the weight of the nodes of T –as determined by a PageRank-inspired technique–, and $f_p(T)$ is a function of the weight of the edges of T. XRANK [26], which ranks result-trees in XML databases, calculates a PageRank-like score for each element of the tree and combines these scores using an aggregation function like max or *sum*.

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 result-tree T for a query Q relies on:

- Attribute value 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 and also ranks the structure of T.

[3] Reference [9] introduces several variations of this scheme (e.g., the node and edge terms above could be multiplied rather than added).

Note that this single-attribute scoring function can be easily extended to incorporate authority flow (i.e., PageRank-style "link"-based) scores [11, 26].

We now turn to the problem of combining single-attribute scores for a result-tree T into a final score for the tree. Notice that the score for a single node t is defined by viewing t as a 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. For example, an application expert could explicitly define a ranking of the tree structures according to the available semanitc information. A simple definition for *Combine* is:

$$Combine(\mathbf{Score}(A, Q), size(T)) = \frac{\sum_{a_i \in A} Score(a_i, Q)}{size(T)} \tag{II.4}$$

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 in Chapter III can handle any combining function that satisfies the following property:

Definition 2 (Node monotonicity) *A combining function Combine satisfies the* node monotonicity *property if, for every query Q and result-trees T and T' of the same structure such that (i) T consists of nodes $t_1, \ldots, t_n$ while T' consists of nodes $t'_1, \ldots, t'_n$ 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 node can be arbitrary, although in the above discussion we assume that the same formula (Equation II.4) calculates the rank for both a single node and a tree of nodes. All ranking functions for result-trees of which we are aware [6, 32, 9], including the one in Equation II.4, satisfy the tuple-monotonicity property, and hence can be used with the execution algorithms described in Chapter III.

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 result-tree that does not include all query keywords, while result-trees

with all query keywords receive the score determined by *Combine*. In contrast, the OR semantics always assigns a result-tree its score as determined by *Combine*, whether the result-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 result-trees either with AND or with OR semantics. Chapter III outlines the architecture of our query processing system, which efficiently identifies the result-trees with the highest relevance scores for a given query.

II.F Related Work

The IR community has focused over the last few decades on improving the quality of relevance-ranking functions for text document collections [47]. We refer the reader to [51] 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[4] and IBM DB2 Text Information Extender[5] use standard SQL to create full text indexes on text attributes of relations. Microsoft SQL Server 2000[6] 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 result-trees for a query.

Goldman et al. [22] tackle the single-object problem in a way similar

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

to how we tackle the connection-as-result problem. 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*. In terms of our framework, this is equivalent to finding all the result-trees and outputting the node of interest of the smallest result-tree.

DBXplorer [6], DISCOVER [32] and BANKS [9] define the result of the connection-as-result problem the same way we do. Their ranking functions have been presented in Section II.E. 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.

Keyword search over XML databases has also attracted interest recently [21, 33, 26]. Florescu et al. [21] 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. Finally, XRANK [26] 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 [11] off-line. These scores are query independent and do not incorporate IR-style keyword relevance.

Chapter III

Efficient Execution

In this chapter we focus on relational databases, although the same techniques are applicable to any labeled graph database, after it is stored in relational format (Chapter V show how to store XML data in relational format). Hence, we use the relational terminology (i.e., "joining tree of tuples" instead of "result-tree", "tuple" instead of "node", "relation" instead of "schema node", edge $u \rightarrow v$ in data graph corresponds to primary to foreign key relationship from u to v) as described in Chapter II. Furthermore, this chapter focuses on the connection-as-result problem where the ranking is done according to two factors: the IR scores of the attribute values of the result-tree, and the structure (schema) of the result-tree. A key assumption we make in this chapter is that the system is a middleware working on top of an already operational database system. Chapters V and VI exploit precomputation opportunities to boost the performance of the system.

Section III.A presents the high-level architecture of the system. Then Section III.B describes the Candidate Network generator algorithm. Sections III.C, III.D present execution algorithms to retrieve all and the top-k results respectively. Section III.E shows a detailed experimental evaluation of the system. Finally, Section III.F discusses about related work.

III.A Architecture

The architecture of our query processing system relies whenever possible on existing, unmodified RDBMS components. Specifically, our architecture (Figure I.7) consists of the following modules:

III.A.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*.[1]

When a query Q arrives, the *IR Engine* uses the *IR Index* to extract from each relation R the *tuple set* $R^Q = \{t \in R \mid \mathit{Score}(t, Q) > 0\}$, 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 $\mathit{Score}(t, Q)$, as required by the top-k query processing algorithms described below.

III.A.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 an integer M, which specifies the maximum size of the joining trees of tuples to be produced as 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.

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

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 c_1: 0.33 c_2: 0.33	p_1: 1 p_2: 1	$c_3 \leftarrow p_2$: 1.17 $c_1 \leftarrow p_1$: 0.67 $c_2 \leftarrow p_2$: 0.67		$c_3 \leftarrow p_2$ $\rightarrow c_2$: 1.11

Figure III.1: CN results for the *Complaints* database and query [Maxtor Netvista], where C stands for *Complaints*, P for *Products*, and U for *Customers*.

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 I.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 details of the CN Generator are described in Section III.B.

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 I.7. Also notice that in this section and in Figure I.7 we use the modified CN generator [30], as we explain in Section III.B.

III.A.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 III.1 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.

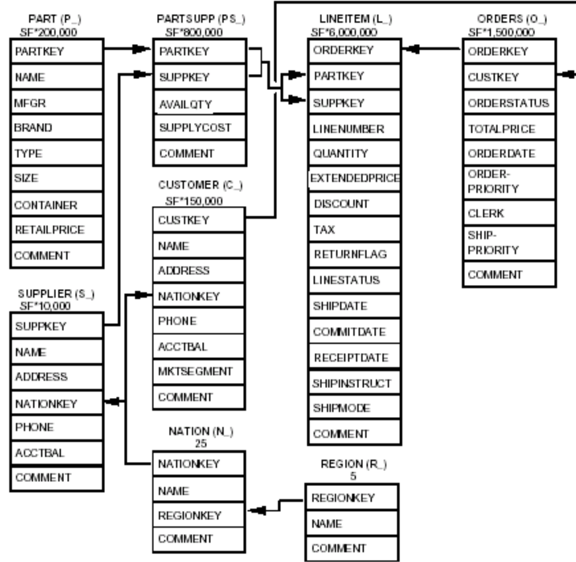


Figure III.2: The TPC-H schema (copied from www.tpc.org)

III.B Candidate Network Generator

There are two versions of the CN Generator. The original version [32] focused on the AND-semantics problem and thus guaranteed that any result-tree produced by any of the CNs contains all keywords. To do so, a new tuple set R_i^K is created for every combination K of the keywords $w_1 \ldots w_m$ and relation R_i. On the other hand, the second version [30] of the CN Generator is more appropriate for OR-semantics, although it can also be used for AND-semantics if an extra post-filtering step is performed to check if every generated joining tree of tuples contains all keywords. In this version, a single tuple set R_i^Q is created for every relation R_i which contains all tuples of R_i that contain any of the keywords of query $Q = [w1, .., w_m]$, that is, $R_i^Q = \bigcup_{j \in 1 \ldots m} R_i^{w_j}$. We first present in detail the algorithm for the first version and then we discuss the differences of the second version. As a running example in this section we use the sample TPC-H[2] database of Figure III.3, which conforms to the schema of Figure III.2.

The Candidate Network Generator inputs the set of keywords $w_1, \ldots, w_m$,

[2]www.tpc.org

ORDERS

	ORDERKEY	CUSTKEY	ORDERSTATUS	TOTALPRICE	ORDERDATE	ORDERPRIORITY	CLERK	...
o_1	1000105	12312	complete	$5,000	5/2/2001	High	John Smith	
o_2	1000111	12312	in process	$3,000	5/1/2001	High	Mike Miller	
o_3	1000125	10001	in process	$7,000	5/1/2001	Low	Mike Miller	
o_4	1000110	10002	complete	$8,000	4/25/2001	Low	Keith Brown	

CUSTOMER

	CUSTKEY	NAME	ADDRESS	NATIONKEY	PHONE	...
c_1	12312	Brad Lou	3811 State Drive, Los Angeles	01	454-1234567	
c_2	10001	George Walters	4365 5th Ave, New York	01	561-2345678	
c_3	10013	John Roberts	3234 Broadway St, San Francisco	01	643-5473921	

NATION

	NATIONKEY	NAME	REGIONKEY	COMMENT
n_1	01	USA	N.America	null

LINEITEM

	ORDERKEY	PARTKEY	SUPPKEY	LINENUMBER	...
l_1	1000105	1122	111222	2	
l_2	1000110	1122	111222	4	
l_3	1000110	2233	222333	3	
l_4	1000111	2233	222333	2	

PARTSUPP

	PARTKEY	SUPPKEY	AVAILQTY	...
p_1	1122	111222	1000	
p_2	2233	222333	400	

Figure III.3: Sample TPC-H database instance

the non-empty tuple sets R_i^w and the maximum CNs' size M and outputs a complete and non-redundant set of CNs. The key challenge is to avoid the generation of redundant *joining networks of tuple sets*, which are defined below.

Definition 3 (Joining Network of Tuple Sets) *A joining network of tuple sets J is a tree of tuple sets where for each pair of adjacent tuple sets R_i^w, R_j^v in J there is an an edge (R_i, R_j) in the schema graph G.*

We show that the CN generation algorithm presented below is (i) complete, ie., every joining tree of tuples is produced by a CN output by the algorithm, and (ii) it does not produce any redundant CNs. Finally we give an example of the algorithm's execution steps.

We must ensure that the joining networks T of tuples that belong to a CN are *total* (i.e., contain all keywords) and also are *minimal*, that is, there is no tuple t with $Score(t, Q) = 0$ that can be removed without breaking the connectivity of T. The condition that a joining network of tuple sets J must satisfy in order to ensure that the produced joining networks of tuples $j \in J$ are total is:

$$\forall w \in \{w_1, \ldots, w_m\}, \exists R_i^K \in J, w \in K \quad \text{(III.1)}$$

For example $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{\{\}}$ is not total with respect to the keyword query "Smith, Miller". Equation III.1 does not ensure minimality. There are two cases when a joining network of tuples T is not minimal.

1. A joining network of tuples T is not minimal if it has a tuple with no keywords as a leaf. In this case we can simply remove this leaf. We carry this condition to joining networks of tuple sets by not allowing free tuple sets as leaves. For example
$ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller} \bowtie CUSTOMER^{\{\}}$
is rejected since it has the free tuple set $CUSTOMER^{\{\}}$ as a leaf.

2. T is not minimal if it contains the same tuple t twice. In this case we can collapse the two occurrences of t. We carry this condition to joining networks of tuple sets by detecting networks that are bound to produce non-minimal joining networks of tuples, regardless of the database instance. According to this condition, the joining network of tuple sets $J = ORDERS^{Smith} \bowtie LINEITEM^{\{\}} \bowtie ORDERS^{Miller}$ is ruled out because the structure of J ensures that all the produced joining networks of tuples $T = o^S \bowtie l \bowtie o^M$ will contain the same tuple twice. To see this suppose that o^S has primary key $p(o^S)$. It is joined with l, so l has foreign key $f_{ORDERS}(l) = p(o^S)$. l will also join with $o^M \in ORDERS^{Miller}$. So, it is $p(o^M) = f_{ORDERS}(l) = p(o^S)$. Hence $o^M \equiv o^S \equiv o^{M,S}$ and T cannot be minimal.

Theorem 1 presents a criterion that determines when the joining networks of tuples produced by a joining network of tuple sets J have more than one occurrences of a tuple.

Theorem 1 *A joining network of tuples T produced by a joining network of tuple sets J has more than one occurrences of the same tuple for every instance of the database if and only if J contains a subgraph of the form R^K—S^L—R^L, where R, S are relations and there is an edge $R \rightarrow S$ in the schema graph.*

Proof: First we prove that if such a subgraph exists and the edge $R \rightarrow S$ is contained in the schema graph, then all joining networks of tuples T produced by J have more than one occurrences of the same tuple. Due to the isomorphism between T and J, T contains a joining network of tuples $s[r_1, r_2]$, where

$r_1 \in R^K, r_2 \in R^M$ and $s \in S^L$. Then we know from the primary to foreign key relationships that $p(r_1) = f_R(s)$ and $p(r_2) = f_R(s)$. Hence $p(r_1) = p(r_2)$. So $r_1 \equiv r_2$.

Next we prove the inverse: If all joining networks of tuples produced by J have more than one occurrences of the same tuple for every instance of the database then a subgraph R^K—S^L—R^M is part of J and the edge $R \rightarrow S$ is contained in the schema graph. Equivalently, we prove that given J, if such a subgraph does not exist then there is an instance I of the database for which J produces a joining network of tuples T with no multiple occurrences of the same tuple. Recall that J is a tree. We prove inductively on the size of T that we can construct T.

Induction Base: Obviously a joining network of tuples of size 0 (i.e., with no joins) has no multiple occurrences of the same tuple, since it consists of a single tuple.

Induction Hypothesis: We assume that we can construct a joining network of tuples $T_n \in J_n$, where J_n is a subtree of J of size n, such that:

$$\forall R^K, R^M \in J_n, \forall t_i \in R^K, \forall t_j \in R^M, t_i, t_j \in T_n \Rightarrow p(t_i) \neq p(t_j)$$

That is, for every pair of tuples in T_n that belong to the same relation R, the tuples have different primary keys.

Induction Step: Now we prove that we can construct $T_{n+1} \in J_{n+1}$, such that the above condition holds for T_{n+1}. If the new tuple r belongs to the new tuple set $R^K \in J_{n+1}$ that is adjacent to $S^L \in J_n$ and tuple $s \in S^L$ and $s \in T_n$, we have two cases:

- If the schema graph contains the edge $R \leftarrow S$, then we just assign a fresh value to $p(r)$ and also set $f_S(r) = p(s)$. (Recall, we have to prove that there is an instance where there are no multiple occurrences. So, we are free to choose the value of r in this instance.)

- If the schema graph contains the edge $R \rightarrow S$, then we have to set $p(r) = f_S(s)$. Hence, if there is another tuple $r' \in T_n$ and $r' \in R^M$ with $p(r) = p(r')$ it must also be $p(r') = f_S(s)$. Given that the subgraph R^K—S^L—R^M does

not exist, the only way that $f_S(s)$ could propagate to $p(r')$ is to have a chain of tuple sets $R^K \sim R_1^{K_1} \sim \cdots \sim R_n^{K_n} \sim R^M \in J_n$, where the inner tuple sets of the chain have the same single attribute as both a foreign and a primary key. This is not allowed, since we have assumed that no set of attributes of any relation is both a primary and a foreign key for two other relations. Hence, $p(r) \neq p(r')$.

□

Hence, we conclude to the following criterion.

Criterion 1 (Pruning Condition) *A CN does not contain a subtree of the form R^K—S^L—R^L, where R and S are relations and the schema graph has an edge $R \rightarrow S$.*

III.B.1 Candidate Networks Generation Algorithm

The candidate network generation algorithm is shown in Figure III.4. First, we create the *tuple set graph* G_{TS}. A node R_i^K is created for each non-empty tuple set R_i^K, including the free tuple sets. An edge $R_i^K \rightarrow R_j^L$ is added if the schema graph G has an edge $R_i \rightarrow R_j$. The algorithm is based on a breadth-first traversal of G_{TS}. We keep a queue Q of "active" joining networks of tuple sets. In each round we pick from Q an active joining network of tuple sets J and either (i) discard J because of the pruning condition (Criterion 1) or (ii) output J as a CN or (iii) expand J into larger joining networks of tuple sets (and place them in Q). We start the traversal from all tuple sets that contain a randomly selected keyword $w_t \in \{w_1, \ldots, w_m\}$.

An active joining network of tuple sets C is expanded according to the following *expansion rule*: A new active joining network of tuple sets is generated for each tuple set R_i^K, adjacent to C in G_{TS}, if either R_i^K is a free tuple set ($K = \{\}$) or after the addition of R_i^K to C every non-free tuple set of C (including R_i^K) contributes at least one keyword that no other non-free tuple set contributes, i.e.,

```
Algorithm CN Generator
Input:   tuple set graph G_TS, M, w_1, ..., w_m
Output:   set of CNs with size up to M
{
  Q: queue of joining networks of tuple sets
  Pick a keyword w_t ∈ {w_1, ..., w_m}
  for each tuple set R_i^K where i = 1, ..., n and w_t ∈ K do
    Add joining networks of tuple sets R_i^K to Q
  while Q not empty do {
    Get head C from Q
    if C satisfies the pruning condition then ignore C
    else if C satisfies the acceptance conditions then output C
    /*There is no reason to extend accepted joining networks of tuple sets*/
    else
      for each tuple set R_i^K adjacent in G_TS (ignoring edge direction) to a node of C
        if (K = {} OR
          ∄R_j^L ∈ (C ∪ R_i^K), L ≠ {} ∧ keywords(C ∪ R_i^K) = keywords((C ∪ R_i^K) − R_j^L))
    /*Expansion rule*/
          and (size of C < M) then {
          if R_i^K is adjacent to R_j^L in C = R_j^L[...] then C ← R_i^K[R_j^L[...]]
          Put C in Q
        }
        else ignore R_i^K
  }  }
```

Figure III.4: Algorithm for generating the candidate networks

$\not\exists R_j^L \in (C \cup R_i^K), L \neq \{\} \wedge keywords(C \cup R_i^K) = keywords((C \cup R_i^K) - R_j^L)$

where $keywords(J)$ returns the union of keywords in the tuple sets of the joining network of tuple sets J, i.e., $keywords(J) = \bigcup_{R_i^K \in J} K$. A free tuple set may be visited more than once. Each non-free tuple set is used at most once in each CN. The reason is that, for all database instances, the result of a joining network of tuple sets J with two occurrences of the same non-free tuple set R_i^K is subsumed by the result of a joining network of tuple sets J', generated by the algorithm, that is identical to J but has $R_i^{\{\}}$ instead of the second occurrence of R_i^K.

In addition, the implementation never places in Q a joining network of tuples sets J that has more than m leaves, where m is the number of keywords in the query. For example, if the keywords are two then only joining paths of tuple sets are placed in Q. Indeed, even if this rule were excluded the output of the algorithm would be the same, since such a network J can neither meet the acceptance conditions listed next nor be expanded into a network J' that meets the acceptance conditions. Nevertheless, the rule leads to cleaner traces and better running time.

The algorithm outputs a joining network of tuple sets J if it satisfies the following *acceptance conditions*:

- The tuple sets of J contain all keywords, i.e., $keywords(J) = \{w_1, \ldots, w_m\}$.
- J does not contain any free tuple sets as leaves.

An important property of the algorithm is that it outputs the CNs with increasing size. That is, the smaller candidate networks, which are the better solutions to the keyword search problem, are output first.

Theorems 2 and 3 prove the completeness and the minimality of the results of the algorithm.

Theorem 2 (Completeness) *Every solution of size M to the keyword query is produced by a CN of size M, output by the CN generator.*

Proof: Suppose a result-tree T of size M is not produced by any CN. Obviously T belongs to a CN C of size M not output by the algorithm. That is, C was either not produced or it was produced and pruned or it did not meet the acceptance conditions.

If C was not produced, then it either violated the expansion rule, so C is subsumed by another candidate network C', or C contains a candidate network C''. Hence T is not minimal since it contains another result-tree.

If C was pruned, then C satisfies the pruning condition. So T contains two tuples $t \equiv t'$ and j is not minimal since we can just collapse t with t'. □

Theorem 3 (No Redundancy) *For each CN C output by the algorithm, given the tuple set graph G_{TS}[3], there is an instance I of the database that produces the same tuple set graph G_{TS}, contains a result joining tree of tuples $T \in C$ and T does not belong to any other CN.*

Proof: The construction of the database instance I proceeds as follows: For each CN C we produce a result-tree $T \in C$ with the property that no tuple $t \in T$ that maps to a free tuple set $TS \in C$ contains any keywords. This last condition ensures that no tuple set that was empty when the CNs were generated will be non-empty in I. Hence for each tuple set $R_i^K \in C$, a tuple $t \in R_i$ is generated that contains all keywords in K. We also make sure that the primary to foreign key relationships hold. We add to I the tuples of each such result-tree.

Now we prove that there is no other CN $C' \not\equiv C$ that produces J if C' is evaluated on I. C' is isomorphic to T, so it is also isomorphic to C. Also, C' contains the same keywords in each tuple set as T does in each tuple, so it also has the same tuple sets with C in the tuple sets that map to each other through the isomorphism, since each set of keywords is contained in exactly one tuple set in a CN (by the expansion rule). Hence $C' \equiv C$. □

[3] *Notice that the CN generator does not examine the tuples of a specific tuple set, but only whether it is empty or not.*

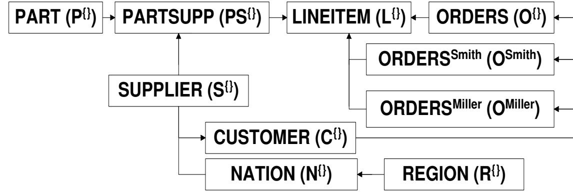


Figure III.5: Tuple set graph for first version of CN generator

Example. We present the execution of the CN generator algorithm for the keyword query "Smith, Miller" on the TPC-H schema and the database instance in Figure III.3, for $M = 6$. That is, we consider candidate networks having at most 5 joins. The tuple set graph is shown in Figure III.5.

Suppose we pick "Smith" as the w_t of the algorithm. Hence we put $ORDERS^{Smith}$ into the queue. The state of the queue and the CNs output in each iteration are shown in Figure III.6. We use the obvious abbreviated names for the relations. Since the query has only two keywords, only joining paths of tuple sets are generated and eventually output.

In the second version [30] of the CN generator, the key difference is that a single tuple set R_i^Q is generated for every relation R_i. Hence the tuple-set graph for the above example would be the one shown in Figure III.7. The only difference in the algorithm of Figure III.4 is that we replace the expansion condition with "The number of non-free tuple sets in CN C 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 AND-semantics.

III.C All-Results Execution

This section describes an execution algorithm that we found to perform well when we want to retrieve all the result-trees (of size up to M) of a keyword query. We focus on AND-semantics and hence use the original CN generator as described in Section III.B. Also, since we output all results, the efficient ranking

#	*Queue/from/candidate networks output*
1a	O^{Smith}
2a	$O^{Smith} \bowtie L^{\{\}}$/1a
b	$O^{Smith} \bowtie C^{\{\}}$/1a
3a	$O^{Smith} \bowtie L^{\{\}} \bowtie O^{\{\}}$(pruned)/2a
b	$O^{Smith} \bowtie L^{\{\}} \bowtie O^{Miller}$(pruned)/2a
c	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}}$/2a
d	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{\{\}}$/2b
e	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{Miller}$/2b
f	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}}$/2b
4a	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie P^{\{\}}$/3c/ $O^{Smith} \bowtie C^{\{\}} \bowtie O^{Miller}$
b	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}}$/3c
c	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{\{\}} \bowtie C^{\{\}}$(pruned)/3d
d	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}}$/3f
	...
5a	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie P^{\{\}} \bowtie PS^{\{\}}$(pruned)/4a
b	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}} \bowtie O^{Miller}$/4b
c	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{Miller}$ /4d
d	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{\{\}}$/4d
e	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie N^{\{\}}$(pruned)/4d
	...
6a	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{\{\}} \bowtie C^{\{\}}$(pruned)/5d/ $O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{Miller}$
	.../ /$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}} \bowtie O^{Miller}$
7	...

Figure III.6: Example

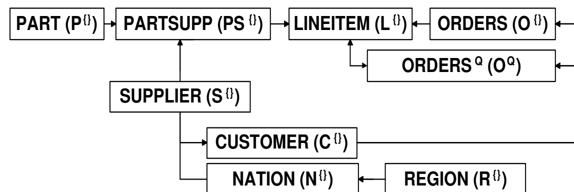


Figure III.7: Tuple set graph for second version of CN generator

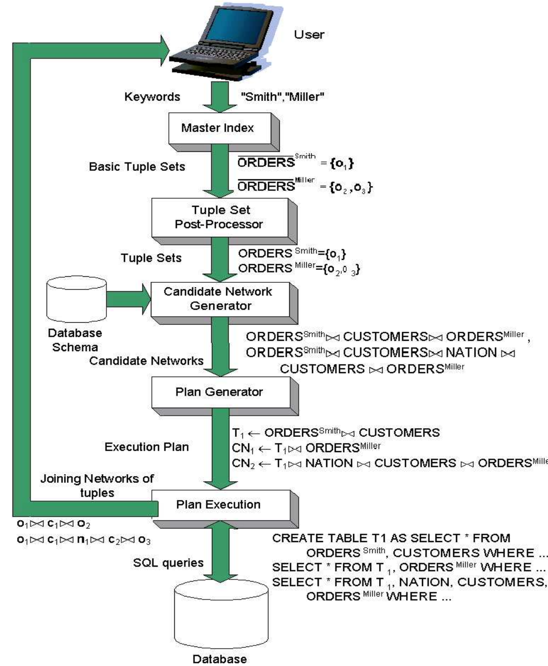


Figure III.8: Architecture for *all results* execution

problem assumes secondary importance (in contrast to the top-k problem discussed in Section III.D). Hence, to simplify this section and focus on the plan generation problem (see below), we rank the results according to their size, although this approach could easily be adapted to more complex ranking functions.

Architecture. The architecture (Figure III.8) is more detailed than the one described in Section III.A to show two unique stages neccesary for this setting.

First, since we use the original CN generation algorithm, we need a *Tuple Set Post-Processor* to creates a tuple set R_i^K for every relation R_i and every combination K of the keywords $w_1, \ldots, w_m$. In particular, the *Master Index* inputs a set of keywords $w_1, \ldots, w_m$ and outputs a set of *basic tuple sets* $\bar{R}_i^{w_j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. The basic tuple set $\bar{R}_i^{w_j}$ consists of all tuples of relation R_i that contain the keyword k_j. The master index has been implemented using the Oracle9i InterMedia Text extension, which builds full text indices on single attributes of relations. Then the master index inspects the index of each

attribute and combines the results.

Then the *Tuple Set Post-Processor* takes the basic tuple sets and produces tuple sets R_i^K for all subsets K of $\{w_1, \ldots, w_m\}$, where

$$R_i^K = \{t | t \in R_i \wedge \forall w \in K, t \textit{ contains } w \wedge \\ \forall w \in \{w_1, \ldots, w_m\} - K, t \textit{ does not contain } w\} \quad \text{(III.2)}$$

i.e., R_i^K contains the tuples of R_i that contain all keywords of K and no other keywords.

The tuple sets are obtained from the basic tuple sets using the following formula.

$$R_i^K = \bigcap_{w \in K} \bar{R}_i^w - \bigcup_{k \in \{w_1, \ldots, w_m\} - K} \bar{R}_i^w \quad \text{(III.3)}$$

Second, before the CNs are input to the execution module, they are processed by the *Plan Generator*, which optimizes their evaluation. This optimization is the key contribution of this section.

Definition 4 (Execution Plan) *Given a set* $C_1, \ldots, C_r$ *of CNs, an execution plan is a list* $A_1, \ldots, A_s$ *of assignments of the form* $H_i \leftarrow B_{i_1} \bowtie \cdots \bowtie B_{i_t}$ *where:*

- *Each* B_{i_j} *is either a tuple set or an intermediate result defined in a previous assignment. The latter requires that there is an index* $k < i$*, such that* $H_k \equiv B_{i_j}$.
- *For each candidate network* C *there is an assignment* A_i*, that computes* C.

For example, an execution plan for the keyword query shown in Figure III.8 is

$T_1 \leftarrow ORDERS^{Smith} \bowtie CUSTOMER^{\{\}}$,

$C_1 \leftarrow T_1 \bowtie ORDERS^{Miller}$,

$C_2 \leftarrow T_1 \bowtie NATION^{\{\}} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller}$

where T_1 is an intermediate result. The number of joins of this plan is 5, whereas the number of joins to evaluate the two candidate networks without building any

intermediate results would be 6. As the number of CNs increases the difference in the number of joins increases dramatically.

Finally the execution plan is passed to the *Plan Execution* module, which translates the assignments of the plan to SQL statements. The assignments that build intermediate results are translated to "CREATE TABLE" statements and the candidate network evaluation assignments to "SELECT-FROM-WHERE" statements. The union of the results of these "SELECT-FROM-WHERE" statements is the result of the keyword search and it is returned to the user.

Execution The *Plan Generator* module inputs a set of CNs and creates an execution plan to evaluate them as defined in Definition 4. The key optimization opportunity is that typically the CNs share join subexpressions. Efficient execution plans store the common join expressions as intermediate results and reuse them in evaluating the CNs. For example, in Figure III.8 we calculate and store the join expression $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}}$. $CUSTOMER^{\{\}} \bowtie ORDERS^{Miller}$ is also a common join expression but it will not help to store both, as we explain below.

The space of execution plans that can be generated for a set of candidate networks is huge. We prune it by the following two assumptions: First, we define every non-free tuple set to be a *small* relation, since its tuples are restricted to contain specific keywords. The result of a join that involves a small relation is also a small relation. Those assumptions lead to the conclusion that every join expression of the plan must contain a small relation and, hence, all intermediate results are small. Note that both the assumptions and the conclusion follow directly the Wong-Yousefi algorithm ([53]) of INGRES. Indeed, in practice, the intermediate results are sufficiently small to be stored in main memory as we discuss in Section III.E.

Second, the plan generator only considers plans where the right hand side of the assignments $H_i \leftarrow B_{i_1} \bowtie \ldots \bowtie B_{i_t}$ of Definition 4 are joins of exactly two

arguments, i.e., $t = 2$. This policy is based on the assumption that the cost of calculating and storing the results of both $A \bowtie B$ and $A \bowtie B \bowtie C$ is essentially the same with the cost for just calculating and storing the result of $A \bowtie B \bowtie C$, if the DBMS optimizer selects to first calculate $A \bowtie B$ and then the result of $A \bowtie B \bowtie C$. Hence we can store and possibly reuse later $A \bowtie B$ “for free”.

This assumption is very precise when there are indices on the primary and foreign key attributes. Then the joins (and, in particular, the most expensive ones) are executed in a series of index-based 2-way joins. The assumption always held for the Oracle 8i DBMS that we used in our TPC-H-based experimentation. (The assumption deviates from reality when there are no indices and the database chooses multi-way merge-sort joins.)

In summary, the plan generator considers and evaluates the space of plans where the joins have exactly two arguments. Note that once a plan P is selected from the restricted space we outlined, the plan generator eliminates non-reused intermediate results by inlining their definition into the single point where they are used. That is, given two assignments

$$T \leftarrow A \bowtie B$$

$$T' \leftarrow T \bowtie C$$

if T is not used at any other place than the computation of T', the two assignments will be merged into

$$T' \leftarrow A \bowtie B \bowtie C$$

Cost Model. The theoretical study of the complexity of selecting the optimal execution plan is based on a simple cost model of execution plans: We assign a cost of 1 to each join. We use this theoretical cost model in proving that the selection of the optimal execution plan is NP-complete (Theorem 4). It is easy to see that the problem is also NP-hard for the actual cost model described below.

The actual cost model exploits the fact that we can get the sizes of the non-free tuple sets from the master index. We also assume that we know the

selectivity of the primary to foreign key joins, which can be calculated from the sizes of the relations. The actual cost model defines the cost of a join to be the size of its result in number of tuples. (The cost model can easily be modified to work for the size in bytes instead of the number of tuples.) The cost of the execution plan is the sum of the costs of its joins. Notice that since there are indices on the primary and foreign keys, the cost of a join is proportional to the size of its result, since the joins will typically be index-based joins.

The problem of deciding which intermediate results to build and store can be formalized as follows:

Problem 1 (Choice of intermediate results) *Given a set of CNs, find the intermediate results that should be built, so that the overall cost of building these results and evaluating the CNs is minimum.*

Theorem 4 shows that Problem 1 is NP-complete on the sum of the sizes of the CNs with respect to the theoretical cost model defined above.

Theorem 4 *Problem 1 is NP-complete.*

Proof Sketch: To prove that Problem 1 is NP-complete we need to reduce in polynomial time a known NP-complete problem to it. We selected the problem of data compression via textual substitution ([52]), where a source string needs to be compressed by replacing common substrings with corresponding symbols from a dictionary. Specifically, we considered the case of the external macro model, where the dictionary is stored separately from the encoded data. The source data is treated as a finite string over some alphabet. A source string is encoded as a pair of strings, a dictionary and a skeleton. The skeleton contains characters of the input alphabet interspersed with pointers to substrings of the dictionary. The dictionary is also allowed to contain pointers to substrings of the dictionary. The source string is recovered by substituting dictionary strings for pointers. As an example consider the string $w = aaBccDaacEaccFacac$, which might be encoded under the external macro model as $x = aacc\#(1,2)B(3,2)D(1,3)E(2,3)F(2,2)(2,2)$, where

separates the dictionary from the skeleton and a pointer is denoted as a pair (n, m), where n indicates the position of the first character in the target and m indicates the length of the target. In the data compression problem, the goal is to minimize the size of the encoded string of the source string. In [52], it is proven that this problem is NP-complete.

The mapping between the two problems goes as follows: The dictionary represents the intermediate results in Problem 1 and the data map to the candidate networks. We assume that there is no overlap between the pointers to the dictionary, that is, their targets do not overlap. We need this assumption because for example we do not necessarily know the result of $A \bowtie B$ if we have stored the result of $A \bowtie B \bowtie C$. Also, the scheme is recursive. That is, a macro body (i.e., a string that is a target of a pointer) is allowed to itself contain pointers. We need this, because for example $A \bowtie B \bowtie C$ may use the result of $A \bowtie B$. □

III.C.1 Greedy algorithm

Figure III.9 shows a greedy algorithm that produces a near-optimal execution plan, with respect to the actual cost model defined above, for a set of CNs by choosing in each step the join m between two tuple sets or intermediate results that maximizes the quantity $\frac{frequency^a}{log^b(size)}$, where $frequency$ is the number of occurrences of m in the CNs, $size$ is the estimated number of tuples of m and a, b are constants. The $frequency^a$ term of the quantity maximizes the reusability of the intermediate results, while the $log^b(size)$ term minimizes the size of the intermediate results that are computed first. We have experimented (Section III.E) with multiple combinations of values for a and b and found that the optimal solution is closer approximated for $\{a, b\} = \{1, 0\}$, when the size of the CNs (and the reusability) increases.

We perform a worst case time analysis of the greedy algorithm. The while loop is executed at most $|S| \cdot M$ times if every join has a frequency of 1, where $|S|$ is the number of candidate networks. The calculation of Z takes time $|S| \cdot M$. We

```
Algorithm Select list of intermediate results
Input: set S of CNs of size ≤ M
Output: list L of intermediate results to build
{
  while not all CNs in S have been added to L do {
    Let Z be the set of all small join subexpressions of 1 join
      contained in at least one CN in S;
    Add the intermediate result m with the maximum frequency^a / log^b(size) value in Z to L;
    Rewrite all CNs in S to use m where possible;
  }
}
```

Figure III.9: Greedy algorithm for selecting a list of intermediate results to build

assume that we traverse a candidate network of size M_1 in time $O(M_1)$. In each step, we keep a hash table H with each intermediate result in Z and its frequency. Hence we check if an intermediate result is already in H and increase its frequency in $O(1)$. Finding the intermediate result in H that maximizes $\frac{frequency^a}{log^b(size)}$ takes time $|S| \cdot M$. The rewriting step also takes time $|S| \cdot M$. Hence the total execution time takes in the worst case time $O((|S| \cdot M)^2)$.

The greedy algorithm may output a non optimal list of intermediate results. However, in special cases the greedy is guaranteed to produce the optimal plan. One such case is described by the theorem below:

Theorem 5 *The greedy algorithm for* $(a, b) = (1, 0)$ *is optimal for* $m = 2$ *keywords, when each of them is contained in exactly one relation.*

Proof: Keywords w_1, w_2 are contained in relations R_1, R_2 respectively. Hence every CN will be a joining sequence of tuples with tuple sets $R_1^{w_1}, R_2^{w_2}$ as ends. Suppose that the join subexpression $I \bowtie S$ is selected in an iteration of the algorithm, where I is a previously built intermediate result (or R_1 or R_2 in the first step) and S is a tuple set or an intermediate result. Assume WLOG that

$I = R_1^{w_1} \bowtie \cdots \bowtie R$. If $I \bowtie S$ belongs to n_1 CNs and I belongs to n_2 of them ($n_2 \leq n_1$), the benefit (decrease of overall cost) of building $I \bowtie S$ is $n_2 - 1$. Since n_2 decreases as the algorithm progresses, so does the benefit of the selected subexpression. Notice that by building $I \bowtie S$, it is not possible to decrease the benefit of a subexpression that will be selected next except for the benefit of subexpressions of the form $I \bowtie S \bowtie Q$. But the total benefit of all such subexpressions is less or equal to the benefit of $I \bowtie S$ [4], because all candidate networks that contain these subexpressions, also contain $I \bowtie S$. Hence, building $I \bowtie S$ leads to an optimal solution. So the greedy algorithm is optimal. □

III.D Top-k Execution

This section presents algorithms to efficiently retrieve the top-k result-trees of a keyword query on a relational database. These algorithms work for monotone ranking functions (Definition 2) as the one of Equation II.4. Furthermore, the primary focus of this section is OR-semantics (AND-semantics can be implemented by adding a postprocessing step to check if all keywords are contained in the joining trees of tuples) and hence we use the second version of the CN generator (Section III.B). The architecture used is the one described in Section III.A.

The presented algorithms handle the following 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 the query processing algorithm used in DBXplorer [6]. 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

[4] This does not hold for more than 2 keywords, because building $I \bowtie S$ affects the benefit of all subexpressions adjacent to both R and S.

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 III.E.

III.D.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 algorithm of DBXplorer [6] 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.[5] 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.

III.D.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

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

are the most likely to produce high-score tuple trees using the combining function above, are evaluated first. As we discuss in Section III.E, the *Sparse* algorithm is the method of choice for queries that produce relatively few results.

III.D.3 Single Pipelined Algorithm

The *Single Pipelined* algorithm (Figure III.10) 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 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 I.7 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-

```
Single Pipelined Algorithm(C, Q, k, Score(.), TS_1, ..., TS_v){
01. h(TS_i): top unprocessed tuple of TS_i (i.e., not yet added to S(TS_i))
02. S(TS_i): prefix of TS_i retrieved so far; initially empty
03. R: queue for not-yet-output results, by descending Score(T, Q)
04. Execute parameterized query q(h(TS_1), ..., h(TS_v))
05. Add results of q to R
06. Output all results T in R with Score(T, Q) ≥ max_{i=1}^{v} MPFS_i
07. For i = 1, ..., v move h(TS_i) to S(TS_i)
08. While (fewer than k results have been output) do {
09.   Get tuple t = h(TS_M), where MPFS_M = max_{i=1}^{v} MPFS_i
10.   Move t to S(TS_M)
11.   For each combination (t_1, ..., t_{M-1}, t_{M+1}, ..., t_v) of tuples
       where t_i ∈ S(TS_i) do {
12.     Execute parameterized query q(t_1, ..., t_{M-1}, t, t_{M+1}, ..., t_v)
13.     Add results of q to R}
14.   Output all new results T in R with Score(T, Q) ≥ max_{i=1}^{v} MPFS_i}}
```

Figure III.10: The *Single Pipelined* algorithm.

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)*[6] 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 $S(TS_i)$):

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

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 III.10) is safely returned as one of the top-k results if its associated score is no less than $\overline{MPFS}$.

[6]Notice that *MPS*, as defined in Section III.D.2, is equivalent to *MPFS* before the evaluation of the CN begins (i.e., before any parameterized query is executed).

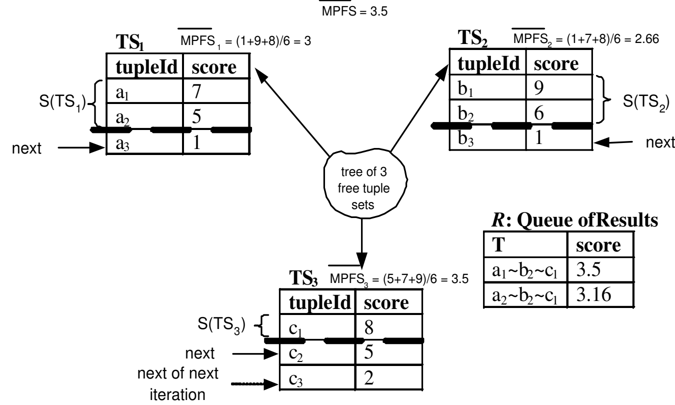


Figure III.11: Snapshot of a *Single Pipelined* execution.

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 III.E show that this choice of tuple set results in better performance over random or round-robin choices.

Example 1 *Figure III.11 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 II.4 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 III.E, 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.

III.D.4 Global Pipelined Algorithm

The *Global Pipelined* algorithm (Figure III.12) 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 [13], 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.

Example 2 *Figure III.13 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.*

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 III.12: The *Global Pipelined* algorithm.

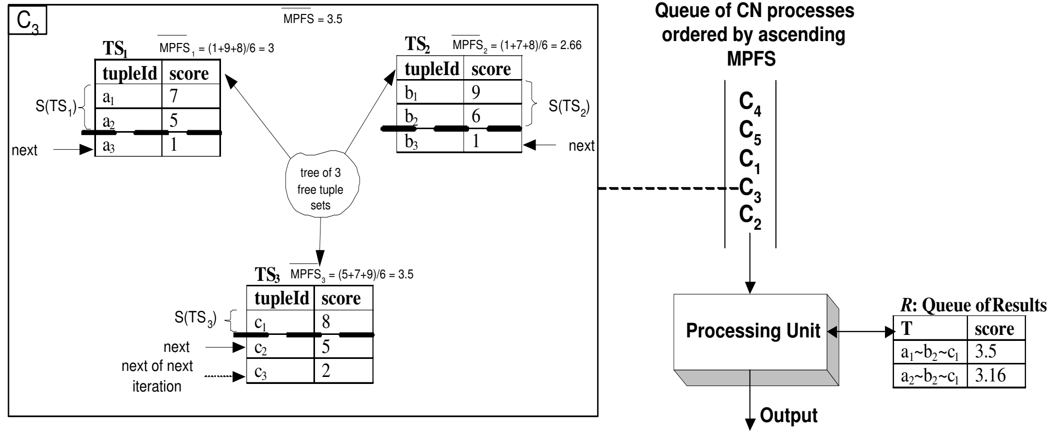


Figure III.13: Snapshot of a *Global Pipelined* execution.

After a minimal portion of the current CN C is evaluated, its $\overline{MPFS}$ is updated, which redefines the priority of C.

Example 3 *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 III.12, 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* $1.165 \geq GMPFS$*. 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.

III.D.5 Hybrid Algorithm

As mentioned briefly above, *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 III.14) 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

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

Figure III.14: The *Hybrid* algorithm.

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 III.E.[7]

III.E Experiments

Section III.E.1 evaluates the pruning capabilities of the original CN generator algorithm presented in Section III.B. Then, Sections III.E.2 and III.E.3 evaluate the algorithms to retrieve all the results and the top-k results of a keyword query respectively.

III.E.1 CN generator

We measure the pruning efficiency of the CN generator. In particular, we measured how many joining networks of tuple sets are ruled out based on the

[7] 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 III.14.

#keyw	*JNTS_K*	*JNTS_L*	*CNs*	*neTS's*
2	25	5.355	4.485	2.96
3	55.22	13.86	9.27	4.35
4	85.69	33.88	24.03	5.91
5	101	37.3	26	7.12

(a) Fix maximum candidate networks' size to 4

MaxCNsize	*JNTS_K*	*JNTS_L*	*CNs*	*neTS's*
2	0.95	0.95	0.95	2.96
3	3.72	2.36	2.12	2.96
4	25	5.355	4.485	2.96
5	422.88	10.36	6.4	2.96
6	6941	24.75	11.45	2.96

(b) Fix number of keywords to 2

MaxCNsize	*JNTS_K*	*JNTS_L*	*CNs*	*neTS's*
2	0.59	0.59	0.59	4.35
3	5.01	3.91	3.35	4.35
4	55.22	13.86	9.27	4.35
5	639.61	50.49	29.51	4.35
6	7532	223	103.66	4.35

(c) Fix number of keywords to 3

Figure III.15: Evaluation of the candidate network generator

pruning conditions of the CN generator. We use the TPC-H schema (Figure III.2) but we do not a the TPC-H dataset in this experiment, because we want to control the distribution of the number of occurrences of each keyword. So, we randomly put the keywords of the keyword query in the relations. Each keyword is contained in a relation R with probability $a \cdot log(size(R))$, where $size(R)$ is the number of tuples in R as defined in the TPC-H specifications for scale factor SF=1. We selected $a = \frac{1}{10 \cdot log_2(6{,}000{,}000)}$. This means that the probability that a keyword is contained in the $LINEITEM$ relation, which is the largest one, is $\frac{1}{10}$, since $size(LINEITEM) = 6,000,000$. This probability is about $\frac{1}{100}$ for the $REGION$ relation, which is the smallest one. We measure three numbers of joining networks of tuple sets for each execution of the experiment.

1. $JNTS_K$ is the number of joining networks of tuple sets of size up to M that have the following properties:
   - They contain all keywords of the keyword query, i.e., they are total.
   - No non-free tuple set can be replaced by a free tuple set and still have all keywords in the joining network of tuple sets.
2. $JNTS_L$ is the number of joining networks of tuple sets that have only non-free tuple sets as leaves in addition to the above properties.
3. CNs is the number of candidate networks generated. Those candidate networks have one more property in addition to the above properties; they do not produce joining networks of tuples with more than one occurrences of the same tuple (Criterion 1).

We also measure the number of non-empty basic tuple sets (*neTS's*) generated in each execution. Figure III.15 shows the average results of the experiment for 1000 executions. Notice that the ratio $\frac{CNs}{JNTS_L}$ decreases as the maximum size of the output candidate networks increases, i.e., Criterion 1 prunes more when the candidate networks are larger. The reason is that the trigger of Criterion 1 has more places to happen in a large candidate network.

III.E.2 All-Results algorithms

We evaluate the all-result algorithms presented in Section III.C with detailed performance evaluation on a TPC-H database. First, we compare the plans produced by the greedy to the ones produced by the optimal, where the optimal execution plan is computed using an exhaustive algorithm. Then, we compare the speedup in runtime performance for generating and executing the execution plan using the greedy and the optimal algorithm compared to the naive method, where no intermediate results are built. Finally, we compare the overall execution times of the system for some typical keyword queries to the naive method and to the optimal method.

We use the TPC-H database to conduct the experiments. The size of the database is 100MB. We use Oracle 9i, running on a Xeon 2.2GHz PC with 1GB of RAM. The system has been implemented in Java and connects to the DBMS through JDBC. The master index is implemented using the full-text Oracle9i interMedia Text extension. The basic tuple set of relation R for keyword w is produced by merging the tuples returned by the full-text index on each attribute of R. We found out that each keyword is contained on the average in 3.5 relations, that is, 3.5 non-empty basic tuple sets are created for each keyword.

The tuple sets and the intermediate results are stored in tables in the KEEP buffer pool of Oracle 9i, which retains objects in memory, thus avoiding I/O operations. We dedicated 70MB to the KEEP buffer pool. The display time is not included in the measured execution time.

The naive method does not produce any intermediate results – it simply executes each candidate network. The execution times for both the naive method and reuse evaluation method, which builds and reuses intermediate results, depend on the status of the cache of the DBMS. In order to eliminate this factor we warm-up the cache before executing the experiments. The warm-up is done by executing the SQL queries corresponding to the candidate networks produced by the candidate network generator. Hence, we are certain that the warm-up does

$\#keyw$	$\frac{Cost(O)}{Cost(G)}$
2	1
3	0.96
4	0.96
5	0.97

(a) Fix CN size to 4

$MaxCN$	$\frac{Cost(O)}{Cost(G)}$
2	1
3	1
4	0.96
5	0.93
6	0.90

(b) Fix # keywords to 3

Figure III.16: Evaluation of the Plans of the Greedy Algorithm

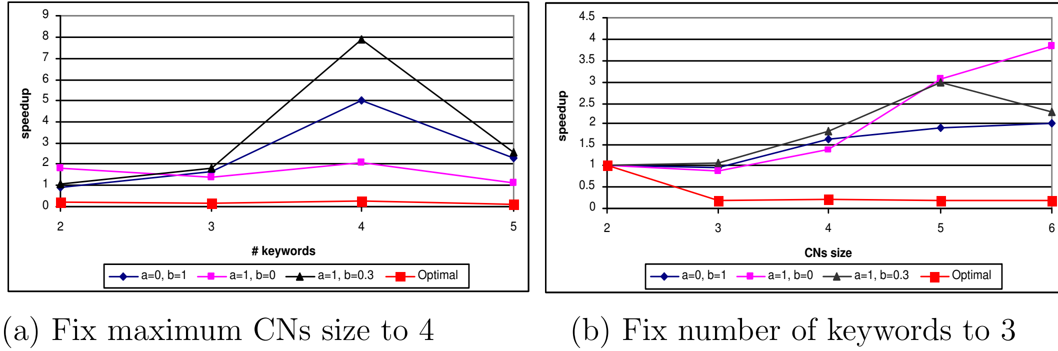
not favor the reuse method more than the naive method.

Quality of Greedy. The quality of the plans produced by the greedy algorithm are very close to the quality of the plans produced by the optimal. We use the same settings with the above experiment. In Figure III.16 we show how well the plans produced by the greedy algorithm perform on the average, compared to the optimal plans for $(a = 1, b = 0)$ for 50 executions. In about 70% of the cases the generated plans turn out to be identical and in the cases where they are different, the differences are fairly small.

Evaluation of Plan Generator. In this experiment we measure the speedup that the system's plan generator induces. In particular, we compare the time spent in the Plan Generator and Plan Execution modules against the baseline provided by the naive method. We also compare the optimal method against the baseline of the naive method. In detail, the measured methods are:

1. *Reuse method.* We calculate the execution plan using the greedy algorithm for three different combinations of values for a, b. In particular, $\{a, b\} \in \{(1, 0), (0, 1), (1, 0.3)\}$. Recall that a and b are the weights we assign to the reusability and the size of the intermediate results respectively.
2. *Naive method.* We evaluate the candidate networks without using any intermediate results.
3. *Optimal method.* We calculate the optimal execution plan using an exhaustive algorithm.

In each execution of the experiment we randomly select m keywords from the



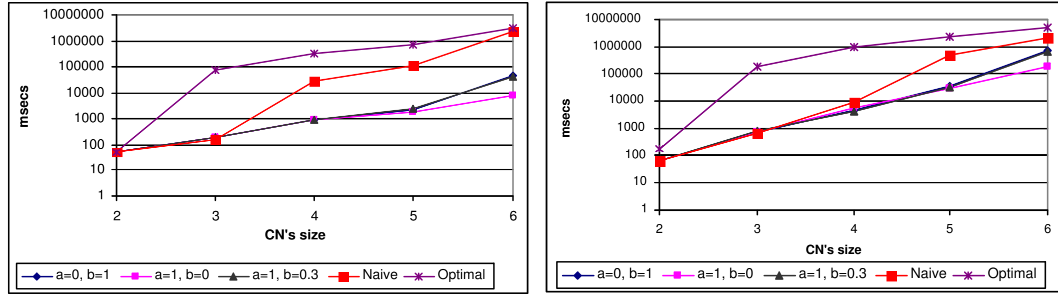
(a) Fix maximum CNs size to 4 (b) Fix number of keywords to 3

Figure III.17: Speedup when using intermediate results

set of words that are contained in the TPC-H database. Then we use the master index to generate the tuple sets and the candidate network generator calculates the candidate networks of size up to M. The execution continues separately for each of the execution methods. We executed the experiment 200 times and measured the average speedup $\frac{Time(Naive)}{Time(other\ method)}$, which indicates that the reuse methods (or the optimal) are $\frac{Time(Naive)}{Time(other\ method)}$ times faster than the naive.

The results are shown in Figure III.17. The optimal method is always worse than the naive due to the great time overhead in discovering the optimal execution plan. Notice in Figure III.17 (a) that the speedup decreases when the number of keywords is greater than 4, because there are more distinct tuple sets in the candidate networks and hence the reusability opportunities decrease since the candidate networks' size is fixed to 4. Also notice in Figure III.17 (b) how the greedy algorithm with $\{a, b\} = \{0, 1\}$ performs better than the one with $\{a, b\} = \{1, 0\}$ when the sizes of the candidate networks are smaller than 5. This happens because the reusability opportunities increase as the size of the candidate networks increases, so the $frequency$ factor of the greedy algorithm becomes dominant. In general, the $(a = 1, b = 0)$ and $(a = 1, b = 0.3)$ options perform better as the difference between the size of the candidate networks and the number of keywords increases, since this creates more opportunities for reusability.

Execution times. Finally, we measure the average absolute execution times to



(a) Fix number of keywords to 2 (b) Fix number of keywords to 3

Figure III.18: Execution times

answer a keyword query using the three methods described above. The execution times in this experiment include the time to generate the candidate networks using the candidate network generator, but not the time to build the tuple sets, which takes from 2 to 4 seconds and could be considerably reduced by using a more efficient master index implementation [6]. The TPC-H dataset is not suitable for this experiment, because it has less than 500 distinct keywords, which are repeated thousands of times. Hence, we inserted into the 100MB TPC-H database, 100 new tuples to each relation. These tuples contain 50 new keywords and each keyword is contained in exactly 50 tuples in two different relations (two non-empty basic tuple sets are created for each keyword). In each execution, the keyword query consists of two randomly selected keywords from the 50 new keywords. Figure III.18 shows the average execution times for the three methods for 100 executions. Again, notice the superiority of the $(a = 1, b = 0)$ and $(a = 1, b = 0.3)$ methods when the size of the candidate networks increases, which happen also to be the toughest cases from a performance point of view. The $a = 1$ parameter leads the greedy to exploit the opportunities for reusing intermediate results.

III.E.3 Top-k algorithms

In this section we experimentally compare the various algorithms described above. For our evaluation, we use the DBLP[8] data set, which we de-

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

composed into relations according to the schema shown in Figure III.19. 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 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)

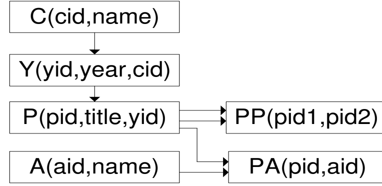


Figure III.19: The DBLP schema graph (C stands for "conference," Y for "conference year," P for "paper," and A for "author").

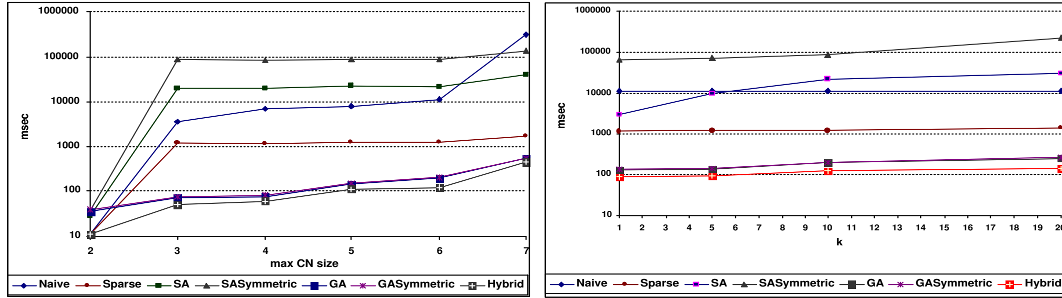
the number m of query keywords. In all the experiments on the *Hybrid* algorithm, we set the tuning constant of Figure III.14 to $c = 6$, which we have empirically found to work well. We compared the following algorithms:

- The *Naive* algorithm, as described in Section III.D.1.
- The *Sparse* algorithm, as described in Section III.D.2.
- The *Single Pipelined* algorithm (*SA*), as described in Section III.D.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 III.D.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 III.D.5.

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

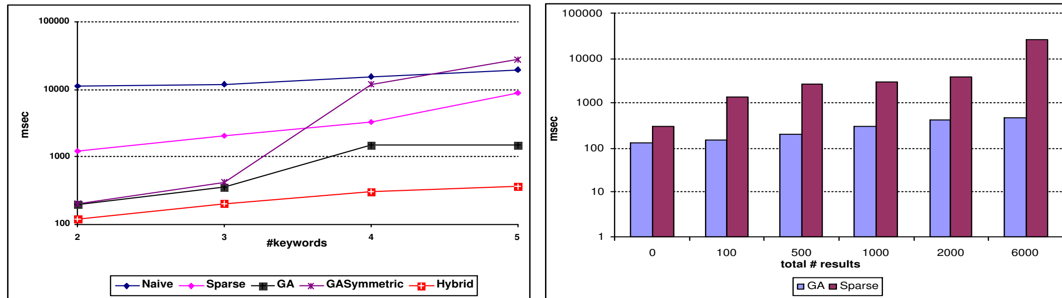
Boolean-OR Semantics

Effect of the maximum allowed CN size. Figure III.20 (a) shows the average query execution time over 100 two-keyword top-10 queries, where each keyword is



(a) Effect of the max CN size (b) Effect of # of objects requested k

Figure III.20: OR semantics

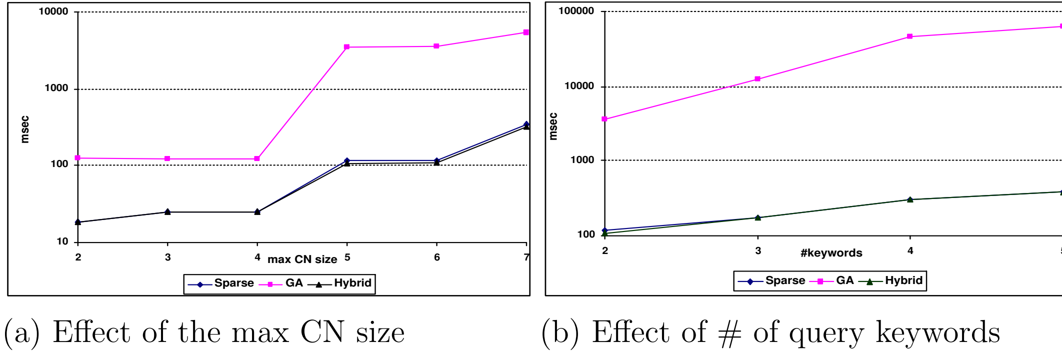


(a) Effect of # of query keywords (b) Effect of the query-result size

Figure III.21: OR semantics

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 III.20 (a)) 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.

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



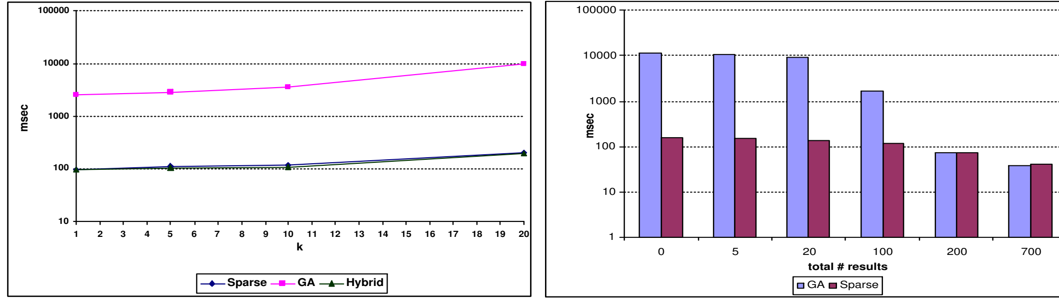
(a) Effect of the max CN size (b) Effect of # of query keywords

Figure III.22: AND semantics

times over 100 queries are shown in Figure III.20 (b). 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 III.21 (a)), 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 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



(a) Effect of # of objects requested, k (b) Effect of the query-result size

Figure III.23: AND semantics

of *GA* and *Sparse* by query-result size. Figure III.21 (b) 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.

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 III.22 (a) ($m = 2$, $k = 10$), III.23 (a) ($m = 2$, $M = 6$), and III.22 (b) ($k = 10$, $M = 6$) 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 III.22 (a) 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 III.19), which is not possible for $M = 4$.

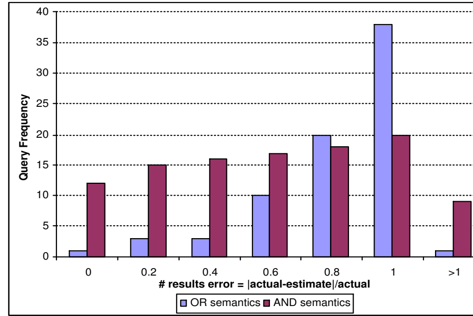


Figure III.24: Quality of the result-size estimates (2-keyword queries; maximum CN size M=6).

Effect of the query-result size. Figure III.23 (b) ($m = 2$, $k = 10$, $M = 6$) shows that, unlike in Figure III.21 (b), 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 III.23 (b) 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 more expensive than executing *Sparse* in the same setting.

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 III.24 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*[9] 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.

III.F Related Work

The work of the Candidate Network Generator reminds of algorithms for answering queries on universal relations [53]. However there are many important differences between universal relations and this work: First, there is the obvious difference that the user of a Universal Relation (UR) needs to know the attributes where the keywords are, in contrast to the user of the system of this work. Second, we create efficient queries that find all connections between the tuples that contain the keywords. In doing so, our system, unlike the UR, has to find connections whose size may not be schema bound and many of them are pruned by the Candidate Network Generator. Finally, in addition to finding the useful connections, we exploit the fact that the connections are "correlated", in the sense that they share join expressions. This leads to a special query optimization algorithm, which is tuned to the specifics of our problem.

One of the criteria that we use to decide that a join expression J is not a candidate network is whether the joining networks of tuples produced by J contain more than one occurrences of the same tuple. Our approach for deciding this property can be viewed as a special case of the chase technique with inclusion dependencies presented in [4]. Our algorithm is simpler, faster and decidable, since it focuses on primary to foreign key relationships.

BANKS answers keyword queries by searching for Steiner trees [44] containing all keywords, using heuristics during the search. Goldman et al. [22] use

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

a related graph-based view of databases. They rank the objects in the *Find* set according to their distance from the objects in the *Near* set, 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 [6] exploits 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. This system relies on a similar architecture, on which we also build in this text (Section III.A). 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 use of common subexpressions by the Plan Generator is a form of multi-query optimization [49, 19, 46]. However the candidate networks have special properties that allow us to develop a more straightforward and efficient algorithm. The first property is that the candidate networks have small relations [53] as leaves, which dramatically prunes the space of useful common subexpressions when applying the Wong-Yusefi algorithm [53]. Second, the candidate networks are not random queries, but share common subexpressions by the nature of their generation as we see in Section III.B. The techniques of [19] cannot be applied in this context since they concentrate on finding common subexpressions as a post-phase to query optimization and our system does not have access to the DBMS optimizer.

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., [43, 17, 31, 12]). However, our problem is unique in that we need to join (ranked) tuples coming from multiple relations in unpredictable ways to produce the final

top-k results.

Natsev et al. [42] extend the work by Fagin et al. [17] 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 III.D.3 can be regarded as an instance of the more general $J*$ algorithm by Natsev et al. [42]. However, $J*$ does not consider predicates over "connecting" relations (i.e., free tuple sets in the terminology of Section III.A). 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). Finally, Ilyas et al. [35] independently developed a top-k algorithm for ranked join queries, which only differs from the Single Pipelined algorithm in the heuristic used to choose the next tuple set from which to retrieve the next tuple.

Chapter IV

Presentation of Results

In Section II.B we defined a result-tree as a subtree of the data graph D that contains all the keywords (AND-semantics). However, the actual presentation of the result-trees was not discussed, as is the case for other relevant works [6, 9, 26] as well. In this chapter we show that the presentation of results is challenging for two reasons: First, a single node of the data graph may not be meaningful by itself to be output if it is not augmented with additional information. We solve this by defining minimum "information units" in the data graph, which we call *target objects* as we explain in Section IV.A. Second, the number of resul-trees for a keyword query is often large due to a form of redundancy as we describe below. This hinders the user from discovering the result he/she desires. To solve this, we propose (Section IV.B) a novel presentation method to allow the user to navogate into the results.

IV.A Minimum Information Units

To ensure that the result of a keyword query is semantically meaningful for the user we introduce the notion of *target objects*. For every node n in the data graph we define (using the schema graph, as we will see later) a segment of the data graph, called *target object* of the node n (or simply called target object when the node n is obvious from the context). Intuitively, a target object of a node n

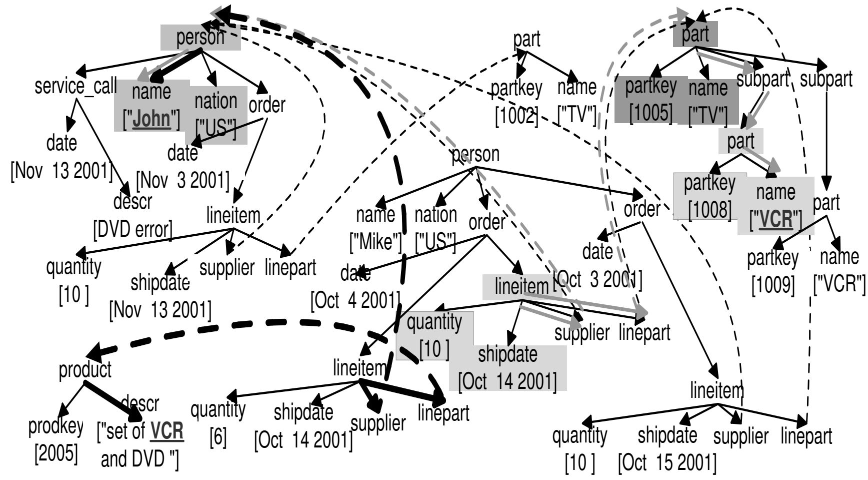


Figure IV.1: Sample XML document

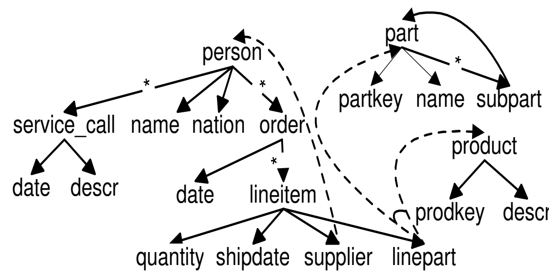


Figure IV.2: TPC-H based schema graph

is a piece of data that is large enough to be meaningful and able to semantically identify the node n while, at the same time, is as small as possible. For example, consider the result-tree N_0 for the keyword query [John VCR] on the data graph of Figure IV.1[1] conforming to the schema graph of Figure IV.2.

$$N_0: \quad \mathit{name}[\mathit{John}] \leftarrow \mathit{person} \leftarrow \mathit{supplier} \leftarrow \mathit{lineitem} \rightarrow$$
$$\mathit{linepart} \rightarrow \mathit{part} \rightarrow \mathit{subpart} \rightarrow \mathit{part} \rightarrow \mathit{name}[\mathit{VCR}]$$

The user would like to know which is the part number of the VCR, which is the part p of which the VCR is a subpart, which line item includes p, and what is the last name of John.[2]

The target objects provide us such information. It makes sense to output the "partkey" of the VCR part as well as the name and "partkey" of the TV. On the other hand it would not make sense to output all the subparts of the TV or the orders of the person. They could be too many and of no interest in semantically identifying the node. Hence, we define the person element with the name and nation subelements to be a target object, and the part with the "partkey" and name to be another target object.

Given a result-tree T with nodes $v_1, \ldots, v_n$ there is a corresponding *target object tree* t,[3] which is a tree whose nodes is a minimal set of target objects $\{t_1, \ldots, t_m\}$ such that for every node $n_k \in T$ there is a $t_l \in t$ such that $\mathit{target}(n_k) = t_l$. There is an edge from a target object t_i to a target object t_j if there is an edge (or as path of *dummy nodes* as defined below) from a node that belongs to t_i to a node that belongs to t_j.

Specification of Target Objects The target objects are defined from an administrator using the *Target Schema Segment (TSS)* graph described next. A *TSS* graph is an uncycled graph whose nodes are called target schema segments. The

[1] This graph is the XML data graph that will be used as a running example in Chapter V.

[2] For simplicity we do not include a last name field in the figures.

[3] The definition does not guarantee the uniqueness of t. The nodes of T may be split in minimal sets of target objects in multiple ways. However, this is of limited practical importance since in practice it is unlikely that target objects overlap with each other in ways that enable a result-tree to be split in multiple ways in target objects.

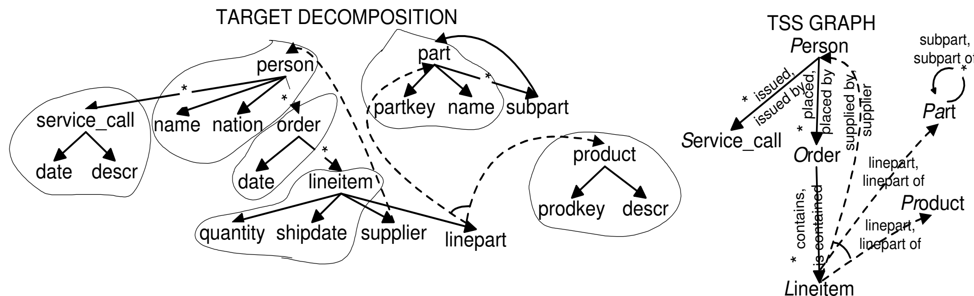


Figure IV.3: Target decomposition of a schema graph

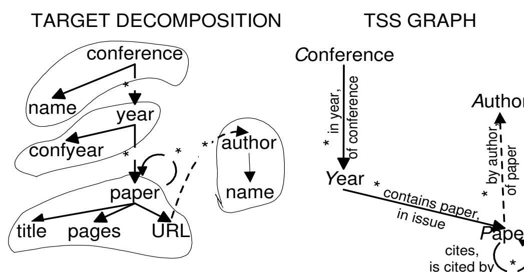


Figure IV.4: Target decomposition of DBLP

TSS graph is derived from a partial mapping of the nodes of the schema graph G. A node t_S is created in G_{TSS} for each set $S = \{s_1, \ldots, s_w\}$ of nodes of G that are mapped to t_S. Some nodes in G, which are called *dummy schema nodes*, are not mapped to any node in G_{TSS}, because they do not carry any information. For example *supplier*, *subpart* and *linepart* are dummy schema nodes. An edge $(t_S, t_{S'})$ is created in G_{TSS} if the schema graph has nodes $s \in S$ and $s' \in S'$, that are connected directly through an edge (s, s') or indirectly through a path of dummy schema nodes. Typically we assign to a node t_S of the TSS graph a name that is the label of the "most representative" schema graph node $s \in S$. For example, the TSS node corresponding to $\{person, name, nation\}$ is named *person* (see Figure IV.3).

Figure V.7 illustrates the TSS graph behind the DBLP demo at www.db.ucsd.edu/XKeyword. Notice the semantic explanations, with the obvious meanings, that annotate the edges. Each edge is annotated with two semantic explanations: the first explains the connection in the direction of the edge and the

second in the reverse direction. Similarly, the semantic explanations of the TPC-H TSS graph are shown in Figure IV.3.

Given the TSS graph, it is straightforward to define a *target decomposition* of the data graph into target objects, connected to each other. For example a target object decomposition of the schema of Figure IV.2 and the corresponding TSS graph are shown in Figure IV.3. The target object tree of the result-tree N_0 presented above is highlighted in Figure IV.1.

IV.B Presentation Graphs

In its simplest result presentation method (Figure I.4 (b)) the XKeyword [33] demo spawns multiple threads, evaluating various plans for producing target object trees, and outputs target object trees as they come. The smaller trees, which are intuitively more important to the user, are usually output first, since they require smaller execution times. The threads fill a queue with target object trees, which are output to the user page by page as in web search engine interfaces.

The naive presentation method described above provides fast response times, but may flood the user with results, many of which are trivial. In particular, as we explained in Section I.E, a redundancy similar to the one observed in multivalued dependencies emerges often. Displaying to the user results involving multivalued dependencies is overwhelming and counter-intuitive. XKeyword faces the problem by providing an interactive interface that allows navigation and hides the trivial results, since it does not display any duplicate information as we show below.

XKeyword's interactive interface presents the results grouped by the candidate networks (see Section III.A.2) they conform to. Intuitively, target object trees that belong to the same candidate network have the same types of target objects and the same type of connections between them. XKeyword groups the results for each candidate network to summarize the different connection types

(schemata) between the keywords and to simplify the visualization of the result.

XKeyword compacts the results' representation and offers a drill-down navigational interface to the user. In particular, a *presentation graph* $PG(C)$ (Figure I.4 (c)) is created for each candidate network C. The presentation graph contains all nodes that participate in some target object tree of C. A sequence of subgraphs $PG_0(C), \ldots, PG_n(C)$ are *active* and are displayed at each point, as a result of the user's actions. The initial subgraph, $PG_0(C)$, is a single, arbitrarily chosen target object tree m of C, as shown in Figure I.4 (c).

An expansion $PG_{i+1}(C)$ of $PG_i(C)$ on a node n of type N is defined as follows. All distinct nodes n', of type N, of every target object tree m' of C are displayed and marked as *expanded* (Figure I.6). Note that we have to consider the statement "of type N" in a restricted sense: A candidate network may involve the same schema type in more than one roles (as is the case with tuple variable aliases in SQL.) For example, in Figure I.6 there are "paper" objects connected to Yannis and "paper" objects connected to Vasilis. We consider those two classes of "paper" objects to be two different types as far as presentation graphs are concerned. In addition a minimal number of nodes of other types are displayed, so that the expanded nodes appear as part of target object trees. More formally, given a presentation graph instance $PG_i(C)$, its expansion $PG_{i+1}(C)$ on a node n of type N has the following properties: (a) $PG_i(C)$ is a subgraph of $PG_{i+1}(C)$, (b) for each target object tree $m' \in C$, where the node $n' \in m'$ is of type N, n' is included in $PG_{i+1}(C)$, (c) for each node $v \in PG_{i+1}(C)$ there is a target object tree z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties and the set of nodes of $PG'_{i+1}(C)$ is subset of the nodes of $PG_{i+1}(C)$.

In the implementation of XKeyword, an expansion on a node n occurs when the user clicks on n. Notice also that if the expanded nodes are too many to fit in the screen then only the first 10 are displayed.

On the other hand, a contraction $PG_{i+1}(C)$ of $PG_i(C)$ on an expanded

node n of type N is defined as follows. All nodes of type N, except for n, are hidden. In addition a minimum number of nodes of types other than N are hidden, while satisfying the restriction that for each node in $PG_{i+1}(C)$ there is a containing target object tree in $PG_{i+1}(C)$ (see condition (c) below). More formally, given a presentation graph instance $PG_i(C)$, its contraction $PG_{i+1}(C)$ on an expanded node n of type N has the following properties: (a) $PG_{i+1}(C)$ is a subgraph of $PG_i(C)$, (b) n is the only node in $PG_{i+1}(C)$ of type N, (c) for each node $v \in PG_{i+1}(C)$ there is a target object tree z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties while $PG'_{i+1}(C)$ has more nodes than $PG_{i+1}(C)$. In the implementation of XKeyword, similar to the expansion, a contraction on an expanded node n occurs when the user clicks on n.

The presentation graphs model allows the user to navigate into the results without being overwhelmed by a huge number of similar target object trees. Furthermore, if he/she is looking for a particular result it is easy to discover it by focusing on one node at a time.

Chapter V

Storage of XML Data

In Chapter III we assumed that we build a middleware system on top of an already operational database. In this chapter we tackle the problem of how to store the data to allow efficient execution. We assume that the original data is in XML format and the underlying storage is a relational DBMS, which is a common setting in recent work [10, 50, 20, 40, 16, 48, 8]. However, the same principles are applicable to other data formats as well.

V.A Architecture

The architecture of this system (Figure V.1) differs from the architecture described in Section III.A, because there is load (preprocessing) stage (in addition to the execution stage), where the XML data is stored into relations. Also, during execution the *Optimizer* is responsible to select the best relations to use from the ones built during the load stage.

Load Stage In the *load stage* the *decomposer* (Section V.B) inputs the schema graph, the TSS graph (see Section IV.A) and the XML (data) graph and creates the following structures:

1. A *master index*, which stores for each keyword w a list of triplets of the form $\langle TO_id, node_id, schema_node\rangle$ where TO_id is the id of the target object

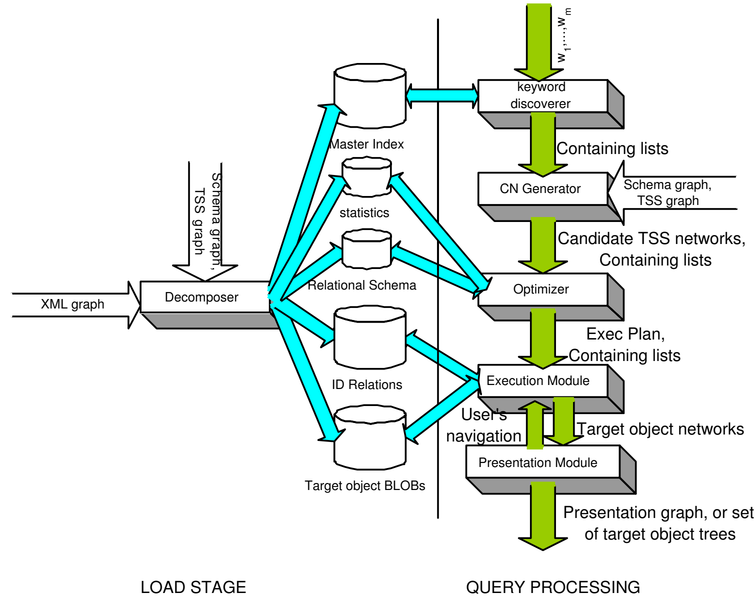


Figure V.1: Architecture

that contains the node of type *schema_node* with id *node_id*, which contains w. The *node_id* [1] and *schema_node* are needed when calculating the score of a candidate network, since as we describe below, the generated relations only store target object id's.

2. A set of statistics specifying: (a) the number $s(S)$ of nodes of type S in the XML graph and (b) the average number $c(S \rightarrow S')$ of children of type S' for a random node of type S.

3. *BLOBs of target objects*, which given an object id instantly return the whole target object.

4. A decomposition of the TSS graph into *fragments*, which correspond to *connection relations* that allow efficient retrieval of target object trees.

Figure V.2 shows a valid decomposition of the TSS graph of Figure IV.3, where the thick arrows and the closed dotted curves denote single edge and multiple edge fragments respectively. We map each fragment into a connection relation.

[1] *node_id* is needed to distinguish two nodes of the same type and of the same target object.

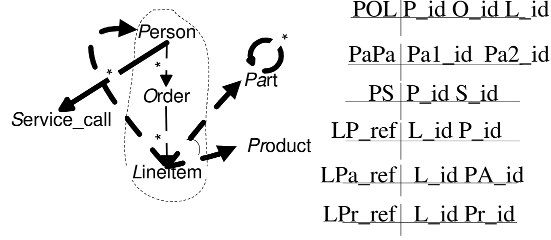


Figure V.2: TSS graph Decomposition

For example, $P \rightarrow O \rightarrow L$ (in short *POL*, since the arrows are unambiguously implied by the TSS graph) is the connection relation that corresponds to the fragment in the dotted line. It stores the connections among the *Person*, *Order* and *Lineitem* TSS's. *LPref* is the connection relation that corresponds to the fragment (indicated by the thick dotted line) containing the reference edge between *Lineitem* and *Person*.

Query Stage The query stage is similar to the one described in Section III.A. A minor difference is that the CN Generator works on the TSS schema graph instead of the schema graph. Hence, instead of candidate networks, it outputs candidate TSS networks. The output candidate TSS networks for $M = 5$ are:

`CTSSN1:` $Part^{TV,name} \rightarrow Part^{VCR,name}$

`CTSSN2:` $Part^{TV,name} \rightarrow Part \rightarrow Part^{VCR,name}$

`CTSSN3:` $Part^{TV,name} \rightarrow Part \rightarrow Part \rightarrow Part^{VCR,name}$

`CTSSN4:` $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part^{VCR,name}$

`CTSSN5:` $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Product^{VCR,descr}$

For example, CTSSN2 corresponds to the CN

$name^{TV} \leftarrow part \rightarrow subpart \rightarrow part \rightarrow subpart$

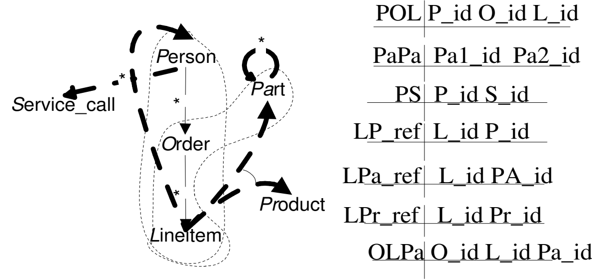


Figure V.3: Another TSS Graph Decomposition

The second difference in this architecture is the *Optimizer* module, which is an adaptation of the Plan Generator of Section III.C which is explained in Section V.C. It uses the schema information on the connection relations and the available statistics to generate the best *Execution Plan* that evaluates the set of candidate TSS networks.

V.B Decompositing XML

The decomposition of the TSS graph into fragments determines how the connections of the XML graph are stored in the database, and consequently the generated execution plan for the candidate TSS networks. We have found that the selected decomposition can dramatically change the performance of the system.

Example 4 *Consider the keyword query [TV VCR] and* $CTSSN4$*:* $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part^{VCR,name}$ *from Section V.A.* $CTSSN4$ *requires three joins given the decomposition of Figure V.2. Consider the TSS graph decomposition of Figure V.3, which includes an* $OLPa$ *fragment. With this decomposition,* $CTSSN4$ *can be evaluated with a single join* $OLP^{TV,part.name} \bowtie OLP^{VCR,part.name}$.

Often we need to build $unfolded$ fragments that contain the same TSS more than once, to store the same edge of the TSS graph more than once, as shown in the example below.

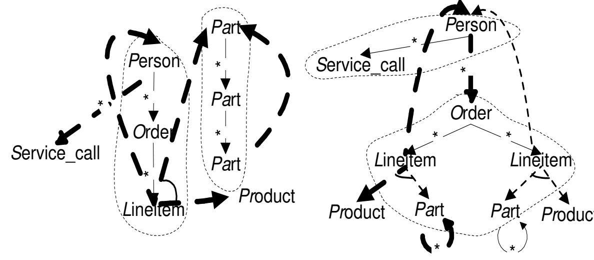


Figure V.4: Unfolded TSS Graph Decompositions

Example 5 *Consider the network $CTSSN2$: $Part^{TV,name} \rightarrow Part \rightarrow Part^{VCR,name}$ of Section V.A. This network connects three Part nodes by following the $Part \rightarrow Part$ edge twice. Under any non-unfolded decomposition this network cannot be executed without a join. However, the first unfolded TSS graph of Figure V.4, which "unrolls" the PartPart cycle, allows the creation of the $Part \rightarrow Part \rightarrow Part$ fragment, which can evaluate $CTSSN2$ without a join.*

Similarly, $CTSSN4$ can be evaluated without a join, if we create the $Part \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part$ fragment on the second unfolded TSS graph of Figure V.4, where the $Order \rightarrow Lineitem$ edge has been "split", i.e., the Order TSS has two children Lineitem TSS's. Notice that not all edges of the unfolded TSS graphs have to be in the decomposition. For example in the second unfolded TSS graph of Figure V.4, the second $Lineitem \rightarrow Person$ edge is not in a fragment, since there is a fragment for the first $Lineitem \rightarrow Person$ edge.

Definition 5 (Walk Set, Unfolded TSS Graph) *A walk set of a TSS graph G, denoted $WS(G)$, is the set of all possible walks in G. A graph G_u is an unfolded TSS graph of the TSS graph G if $WS(G_u) = WS(G)$.*

Definition 6 *TSS Graph Decomposition A decomposition of a TSS graph G is a set of fragments $F_1, \ldots, F_n$, where for each fragment $F\langle N, E\rangle$ there is an unfolded TSS graph G_u of G, such that F is a subgraph of G_u. Every edge of G has to be present in at least one fragment.*

Lemma 1 *Any candidate TSS network can be evaluated given a TSS graph de-*

composition with the properties of Definition 6.

The size of a fragment is the number of edges of the TSS graph that it includes. Note that a TSS graph decomposition is not necessarily a partition of the TSS graph – a TSS may be included in multiple fragments (Figure V.3).

Each fragment $F = \langle N, E \rangle$ corresponds to a *connection relation* R, where each attribute corresponds to a TSS and is of type ID[2]. A tuple is added to R for each subgraph of type F in the *target object graph*, which is the representation of the XML graph in terms of target objects, that is, each node of the target object graph is a target object. Connection relations are a generalization of *path indexes* [18].

V.B.1 Decomposition Tradeoffs

There is a tradeoff between the number of fragments that we build and the performance of the keyword queries, as we show in Section V.E. Assume that we consider solutions to the keyword queries which contain up to M target objects. That is, the maximum size of a candidate TSS network is M. The one extreme is to create the *minimal decomposition*, where a fragment is built for each edge of the TSS graph. Then, each candidate TSS network C requires $S - 2$ joins to be evaluated, where S is the size of C.

The other extreme is the *maximal decomposition*, where a fragment F is built for every possible candidate TSS network C. F is created by replacing the non-free TSS's of C with free TSS's. Then C is evaluated with zero joins. Clearly, the maximal decomposition is not feasible in practice due to the huge amount of space required.

The clustering and indexing of the connection relations are critical because they determine the performance of the joins. In the maximal decomposition, a multi-attribute index is created for every valid (i.e., the keywords can be on these attributes) combination of attributes of every connection relation. In all

[2] In RDBMS's we use the "integer" type to represent the "ID" datatype.

non-maximal decompositions, we found (Section V.E) that the performance is dramatically improved when a connection relation R is clustered on the direction that R is used. For example, consider the execution plan of Section V.A. If the evaluation of $CTSSN3 \leftarrow PaPa^{(TV,part1.name)} \bowtie_{Pa2_id=Pa1_id} PaPa \bowtie_{Pa2_id=Pa1_id} PaPa^{(VCR,part2.name)}$ starts from the left end, then all three $PaPa$ connection relations should be clustered from left to right. If creating all clusterings for each fragment is too expensive with respect to space, then single attribute indices are created on every attribute of the connection relations, since we found that multi-attribute indices are not used by the DBMS optimizer to evaluate join sequences.

The number of joins to evaluate the query Q corresponding to a candidate TSS network is critical, because of the nature of Q, which always starts from "small" connection relations. Also, the connection relations only store ID's and have every single attribute index, which makes the joins index lookups. The significance of the number of joins was verified experimentally (Section V.E). Hence, we specify for each decomposition an upper bound B to the number of joins to evaluate any candidate TSS network of size up to M. For example $B = 0$ and $B = M - 2$ for the maximal and minimal decompositions respectively.

Given B, we generally prefer to build fragments of small sizes to limit the space of storing them. Theorem 6 proves that we can bound the size of the fragments of the decomposition.

Theorem 6 *There is always a decomposition D, whose fragments' maximum size is $L = \lceil \frac{M}{B+1} \rceil$ and any candidate TSS network of size up to M is evaluated with at most B joins.*

Proof: Assume that D is the decomposition that contains exactly all possible fragments of size L. We show how to evaluate a candidate TSS network C of size M (if the size is smaller than M it is an easier case) using D. First we partition the edges of C into connected sets of size L. Notice that the last set s may have size smaller than L. The number of sets is $\lceil M/L \rceil = B + 1$. Each such set corresponds

to a fragment in D. For the last set s we pick a fragment that contains s. Hence we have to join $B+1$ fragments, which needs B joins.

Depending on the TSS graph, we may need to build all possible fragments of size L to satisfy the constraint B on the number of joins. Theorem 7 shows such a class of TSS graphs. □

Theorem 7 *If all edges of the TSS graph are star ("*") edges and* $\exists L \in \mathbf{N}$*, such that* $M = L \cdot (B+1)$*, then the decomposition* D *must contain all fragments of size* L *to satisfy the constraint* B *on the number of joins.*

Proof sketch: Assume that a fragment F of size L is not in D. We show that there is a candidate TSS network C that cannot be evaluated with B joins. C is constructed as follows: If r is the root of F then, we replicate F $B+1$ times and make their root common. Then C needs more than B joins if F is not available. ♢

Often it is not efficient to build all fragments of size L, because a fragment may take up too much space despite its small size (in number of edges). This happens when the corresponding connection relation of a fragment has a non-trivial multivalued dependency (MVD), as the $PaLOLPa$ fragment in Figure V.4, which has the MVD $O_id \twoheadrightarrow L1_id, Pa1_id$. We say that a fragment has an MVD when its corresponding connection relation has an MVD.

Theorem 8 *A fragment* F *has a non-trivial MVD iff* F *contains a path* $p = (e_1, \ldots, e_n)$ *and* $\exists e_i \in \{e_1, \ldots, e_n\}, \exists e_j \in \{e_1, \ldots, e_n\}, i < j$*, and*

- $e_i \in \{\overset{*}{\leftarrow}, \overset{ref}{\longrightarrow}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $e_j \in \{\overset{*}{\rightarrow}, \overset{ref}{\longleftarrow}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $\nexists l, i < l < j-1, e_l \in \{\rightarrow\} \wedge e_{l+1} \in \{\leftarrow\}$

Proof sketch: Assume that R is the corresponding connection relation of F. First we prove that if F contains p, then F has an MVD. We assume that V is the set

of nodes of F. We can assume that there is no star edge $e \in \{e_{i+1}, \ldots, e_{j-1}\}$. If there were, we would consider the path $p' = \{e_i, \ldots, e\}$ or $p' = \{e, \ldots, e_j\}$ if e is $\stackrel{*}{\rightarrow}$ or $\stackrel{*}{\leftarrow}$ respectively. For the same reason we assume that there are no ref edges in $\{e_{i+1}, \ldots, e_{j-1}\}$. Assume that $e_i = (v_i, v_i')$ and $e_j = (v_j, v_j')$. By the hypothesis no l exists, so there is a one-to-one relationship between v_i' and v_j. Also, by the hypothesis it is obvious that one-to-many relatioships exist between v_i' and v_i, and v_j and v_j'. Hence, R has the MVD $v_i' \twoheadrightarrow v_i \cup V_L$, where V_L is the set of nodes of p on the left of v_i.

The inverse specifies that if R has an MVD then the conditions of the theorem hold. Assume that the MVD is $v_m \twoheadrightarrow V_m$, where $v_m \in V$ and $V_m \subseteq V$. If the MVD is non-trivial there must be a one-to-many relationship from v_m to an attribute $v_i \in V_m$ and from v_m to an attribute $v_l' \in (V - V_m - v_m)$. If the hypothesis about l did not hold, then F would be empty since $R{=}\pi_{V_m \cup v_m} R \bowtie_{v_m = v_m} \pi_{V - V_m} R$, by the definition of an MVD. $\diamondsuit$

We classify TSS graph fragments and decompositions based on the storage redundancy in the corresponding connection relations. Connection relations that correspond to a single edge in the TSS graph, by definition are always in 4NF. Some wider connection relations, for example the $OLPa$ relation of Figure V.3 can be in 4NF, however most of them will not be in 4NF. Non-MVD, no-4NF connection relations, are called *inlined* connection relations. A fragment is 4NF, inlined, or MVD, if the resulting connection relation is 4NF, inlined, or MVD respectively.

There are two classes of fragments that should never be built because no candidate TSS network can efficiently use them. We call such fragments *useless*:

1. If a fragment F contains a choice TSS T and more than one children of T, then F is useless, since the children of T can never be connected through T. For example, the fragment $PaLPr$ is useless since $Lineitem$ is a choice TSS.

2. A fragment that contains the construct $T_1 \stackrel{l_1}{\rightarrow} T \stackrel{l_2}{\leftarrow} T_2$ is useless, if $l_1 \neq ref$ and $l_2 \neq ref$, because T_1 and T_2 are never connected through T. For example, the fragment L_1PrL_2 is useless since two $Lineitem$ target objects cannot

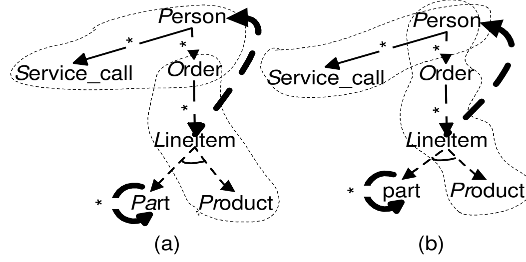


Figure V.5: Replacing an MVD with a non-MVD fragment

connect through a *Part* target object.

We ignore useless fragments in the decomposition algorithm presented below.

Decomposition Algorithm. We use two different decompositions. First, an *inlined, non-MVD* decomposition generated by the algorithm of Figure V.6 is built, where B is the maximum number of joins and M is the maximum candidate TSS network size. This decomposition is used to efficiently generate the top-k results (MTTON's) in the web search engine-like presentation, and the top-1 MTTON of each CN C which corresponds to the initial instance of the presentation graph of C. Second, the minimal decomposition is built, which is used along with the *inlined, non-MVD* decomposition in the on-demand expansion of the presentation graphs. The algorithm in Figure V.6:

- satisfies the B constraint on the number of joins
- avoids building MVD fragments if possible
- builds non-MVD fragments of size larger than $L = \lceil \frac{M}{B+1} \rceil$ if they can eliminate MVD fragments of size L

We say that a candidate TSS network C is *covered* by a decomposition D when C can be evaluated with at most B joins.

Given $M = 5$ and $B = 1$, Figure V.5 shows how the candidate TSS network $S \leftarrow P \rightarrow O \rightarrow L \rightarrow Pr$ is covered if we build the non-MVD fragment $POLPr$ of size $L + 1$ instead of the MVD fragment SPO of size L.

```
Decomposition Algorithm(B,M){
 Add to the decomposition D the non-MVD fragments of size ≤ L;
 Create a list Q of all candidate TSS networks of size up to M not covered by D;
 Add all possible non-MVD fragments of size greater than L, that help in covering at
   least one candidate TSS network C ∈ Q and remove C from Q;
 Add the minimum number of MVD fragments of size up to L
   to cover all candidate TSS networks in Q;
 }
```

Figure V.6: Decomposition Algorithm

V.C Execution Stage Optimizer

The optimizer inputs a set of candidate TSS networks and the containing lists of the keywords. Its goal is to minimize the cost of evaluating the candidate TSS networks by:

- making the best use of the available fragments and
- exploiting reusability opportunities of common subexpression among the candidate TSS networks.

The optimization consists of two stages: (a) rewrite the candidate TSS networks using the available fragments and (b) discover and exploit reusability opportunities among the rewritten candidate TSS networks. The second stage is covered in Section III.C, where heuristics are proposed that use the statistics calculated in the load stage.

Both stages refer to NP-complete problems (Theorems 9, 4). To make things worse, the two stages interact because depending on the rewriting, the reusability opportunities change. Fortunately, it turns out that we can execute the two stages sequentially and still produce an efficient execution plan, because it is rare that an optimal rewriting will reduce the reusability opportunities, because candidate TSS networks which are close enough to share common subexpressions

are usually rewritten using the same set of fragments.

Theorem 9 *The following problem is NP-complete: Given the simple cost model where each join has cost one, and a set of relational views (fragments), rewrite a query (candidate network) using these views to minimize the evaluation cost.*

Proof: See [27]. □

Definition 7 (Rewriting) *A rewriting R of a candidate TSS network N is a graph of fragments, such that:*

- *N is a subgraph of R (containment) and*
- *if any fragment is removed from N, it is no longer a rewriting of R (minimality)*

In a non-redundant decomposition there is exactly one rewriting for each candidate TSS network. The key concerns when rewriting a candidate TSS network N are (a) to exploit the clustering of the selected fragments and (b) to minimize the sizes of the intermediate results. The optimizer calculates the cost of evaluating a rewriting R applying the Wong-Yusefi algorithm and the cost estimations presented in [53]. The evaluation starts from the leaves which are the "small" relations and the join paths meet on one or more "meeting" fragments. The most efficient join method on the meeting fragments is the index join. Notice that the sizes of the non-free TSS's, which are needed in the cost calculation, are calculated from the containing list of the keyword

The number of rewritings is theoretically exponential on the number of fragments, but in practice it is fairly small, given that the size of the candidate TSS networks is bound and the number of fragments that contain each edge of the schema graph is limited. Hence the optimizer can efficiently select an optimal rewriting with respect to the cost model of [53].

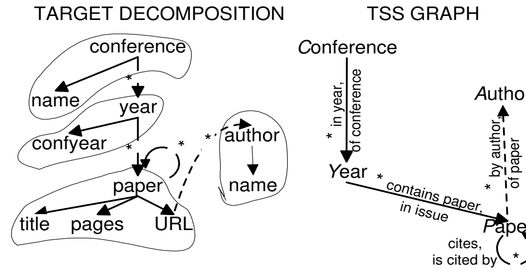


Figure V.7: Target decomposition of DBLP

V.D Related Work

XML data is stored in a relational database [10, 50, 20, 40, 16, 48, 8], to allow the addition of structured querying capabilities in the future and leverage the indexing capabilities of the DBMS's. Some of these works [20, 40, 16] did not assume knowledge of an XML schema. In particular, the Agora project employed a fixed relational schema, which stores a tuple per XML element. This approach is flexible but it is much less competitive than the other approaches, because of the performance problems caused by the large number of joins in SQL queries. This work is different because it exploits the schema information to store the relationships between the target object id's of the XML data. The actual data are stored in XML BLOB's which are introduced in [8].

V.E Experiments

To evaluate the performance of the system we performed a set of experiments. We measure the performance of the keyword queries for various decompositions of the XML schema, for top-k (described in Section III.D) and full results. Notice that we use a simple ranking function [32, 6, 33] that assigns a score $1/size(T)$ to a result-tree T.

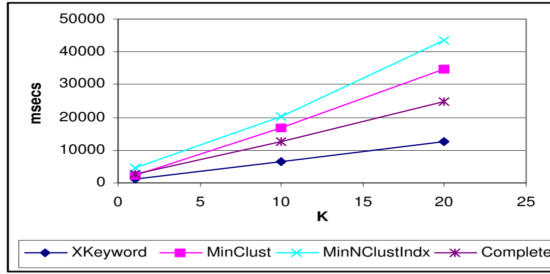
We use the DBLP XML database with the schema shown in Figure V.7. The citations of many papers are not contained in the DBLP database, so we randomly added a set of citations to each such paper, such that the average number

of citations of each paper is 20. We use Oracle 9i, running on a Xeon 2.2GHz PC with 1GB of RAM. XKeyword has been implemented in Java and connects to the underlying DBMS through JDBC. The master index is implemented using the full-text Oracle 9i interMedia Text extension. Clustering is performed using index-organized tables.

Decompositions. We assume that the maximum candidate TSS network size is $M = 7$ and focus on the case of two keywords. Notice that we select a big M value to show the importance of the selected decomposition. The absolute times are an order of magnitude smaller when we reduce M by one. We require that the maximum number of joins is $B = 2$, hence from Theorem 6 it is $L = 3$. We compare five different decompositions:

1. The *XKeyword* decomposition created by the algorithm of Figure V.6.
2. The *Complete* decomposition, which consists of all fragments of size L.
3. The *MinClust* decomposition, which is the minimal decomposition with all possible clusterings for each fragment.
4. The *MinNClustIndx* decomposition, which is the minimal decomposition with single attribute indices on every attribute of the ID relations.
5. The *MinNClustNIndx* decomposition, which is the minimal decomposition with no indices or clustering.

We compare the average performance of these decompositions to output the top-k results for each candidate network. The results are shown in Figure V.8 (a). Notice that the *Complete* decomposition is slower than *MinClust* although it requires a smaller number of joins, because of the huge size of the fragments that correspond to relations with multi-valued dependencies and the more efficient caching performed in the *MinClust* decomposition. Also notice that the non-clustered decompositions (the results for *MinNClustNIndx* are not shown,



(a) Top-k results

Decomposition	secs
$XKeyword$	170.8
$Complete$	302.9
$MinClust$	225.2
$MinNClustIndx$	347.4
$MinNClustNIndx$	137.3

(b) All results

Figure V.8: Execution times

because they are worse by an order of magnitude) perform poorly for the top-k results.

Figure V.8 (b) shows the average execution times to output all the results for each candidate network. Notice that the $MinNClustNIndx$ is the fastest, since the full table scan and the hash join is the fastest way to perform a join when the size of the relations is small relatively to the main memory and the disk transfer rate of the system, which is the case here, since all relations of the minimal decomposition have just two id (integer) attributes.

Chapter VI

ObjectRank

This chapter describes in detail the authority flow ranking factor (Section II.D). In particular, we focus on the single-object problem Section II.B, for which we show that there are efficient ways to calculate the authority flow (ObjectRank [7]) scores. On the other hand, applying the authority flow factor to measure the association between the nodes of result-trees (Section II.C) is more challenging on the performance and semantics level and is left as future work.

Section VI.A presents the basics of the PageRank algorithm. Then, Section VI.B defines the data model and the semantics of the solution to the problem. Section VI.C presents the architecture of ObjectRank, and Section VI.D describes algorithms to calculate the ObjectRank scores. Finally, Sections VI.E and VI.F experimentally evaluate the system and discuss related work respectively.

VI.A Background

We describe next the essentials of PageRank and authority-based search, and the random surfer intuition. Let (V, E) be a graph, with a set of nodes $V = \{v_1, \ldots, v_n\}$ and a set of edges E. A surfer starts from a random node (web page) v_i of V and at each step, he/she follows a hyperlink with probability d or gets bored and jumps to a random node with probability $1 - d$. The PageRank value of v_i is the probability $r(v_i)$ that at a given point in time, the surfer is at v_i.

If we denote by $\mathbf{r}$ the vector $[r(v_1), \ldots, r_(v_i), \ldots, r(v_n)]^T$ then we have

$$\mathbf{r} = d\mathbf{A}\mathbf{r} + \frac{(1-d)}{|V|}\mathbf{e} \tag{VI.1}$$

where $\mathbf{A}$ is a $n \times n$ matrix with $A_{ij} = \frac{1}{OutDeg(v_j)}$ if there is an edge $v_j \rightarrow v_i$ in E and 0 otherwise, where $OutDeg(v_j)$ is the outgoing degree of node v_j. Also, $\mathbf{e} = [1, \ldots, 1]^T$.

The above PageRank equation is typically precomputed before the queries arrive and provides a global, keyword-independent ranking of the pages. Instead of using the whole set of nodes V as the *base set*, i.e., the set of nodes where the surfer jumps when bored, one can use an arbitrary subset S of nodes, hence increasing the authority associated with the nodes of S and the ones most closely associated with them. In particular, we define a *base vector* $\mathbf{s} = [s_0, \ldots, s_i, \ldots, s_n]^T$ where s_i is 1 if $v_i \in S$ and 0 otherwise. The PageRank equation is then

$$\mathbf{r} = d\mathbf{A}\mathbf{r} + \frac{(1-d)}{|S|}\mathbf{s} \tag{VI.2}$$

Regardless of whether one uses Equation VI.1 or Equation VI.2 the PageRank algorithm solves this fixpoint using a simple iterative method, where the values of the (k+1)-th execution are calculated as follows:

$$\mathbf{r}^{(k+1)} = d\mathbf{A}\mathbf{r}^{(k)} + \frac{(1-d)}{|S|}\mathbf{s} \tag{VI.3}$$

The algorithm terminates when $\mathbf{r}$ converges, which is guaranteed to happen under very common conditions [41]. In particular, $\mathbf{A}$ needs to be irreducible (i.e., (V, E) be strongly connected) and aperiodic. The former is true due to the damping factor d, while the latter happens in practice.

The notion of the base set S was suggested in [11] as a way to do personalized rankings, by setting S to be the set of bookmarks of a user. In [29] it was used to perform topic-specific PageRank on the Web. We take it one step further and use the base set to estimate the relevance of a node to a keyword query. In

Parameter property	*Parameters*
`Application - specific`	`authority transfer rates, global ObjectRank calculation, damping factor`
`Combination of scores`	`normalization scheme, global ObjectRank weight, AND or OR semantics`
`Performance`	`epsilon, threshold`

Table VI.1: Parameters of ObjectRank

particular, the base set consists of the nodes that contain the keyword as explained next.

VI.B Data Model and Semantics

In this section we formally define the framework of ObjectRank, and show how ObjectRank ranks the nodes of a database with respect to a given keyword query, given a set of calibrating (adjusting) parameters (Table VI.1). In particular, Section VI.B.1 describes how the database and the authority transfer graph are modeled. Section VI.B.2 shows how the keyword-specific and the global ObjectRanks are calculated and combined to produce the final score of a node. Section VI.B.3 presents and addresses the challenges for multiple-keyword queries. Finally, Section VI.B.4 discusses how the frequency of the keywords in taken into account and Section VI.B.5 compares our approach to HITS [38].

VI.B.1 Data Graph, Schema, and Authority Transfer Graph

The data graph $D(V_D, E_D)$ and the schema graph $G(V_G, E_G)$ have been defined in Section II.A. In this section to simplify, we assume that every node has a single attribute and the label (value) of this attribute is the label (value) of

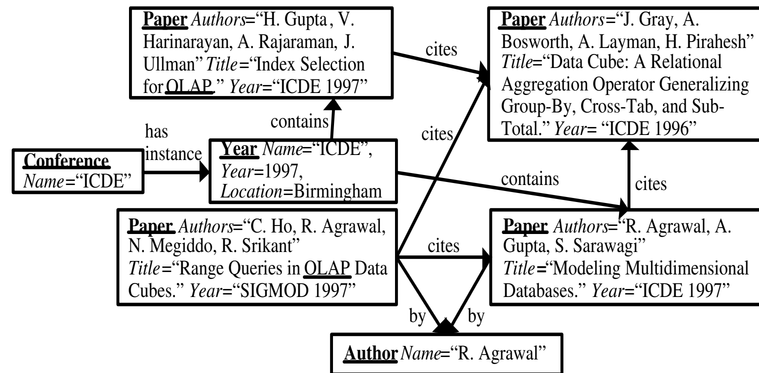


Figure VI.1: A subset of the DBLP graph

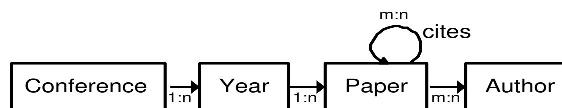


Figure VI.2: The DBLP schema graph.

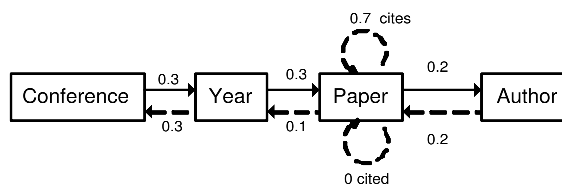


Figure VI.3: The DBLP authority transfer schema graph.

the node. Another (from the example of Section II.A) example of a data and the corresponding schema graph is shown in Figures VI.1 and VI.2.

Authority Transfer Schema Graph. From the schema graph $G(V_G, E_G)$, we create the *authority transfer schema graph* $G^A(V_G, E^A)$ to reflect the authority flow through the edges of the graph. This may be either a trial and error process, until we are satisfied with the quality of the results, or a domain expert's task. In particular, for each edge $e_G = (u \rightarrow v)$ of E_G, two *authority transfer edges*, $e_G^f = (u \rightarrow v)$ and $e_G^b = (v \rightarrow u)$ are created. The two edges carry the label of the schema graph edge and, in addition, each one is annotated with a (potentially different) *authority transfer rate* - $\alpha(e_G^f)$ and $\alpha(e_G^b)$ correspondingly. We say that a data graph conforms to an authority transfer schema graph if it conforms to the corresponding schema graph. (Notice that the authority transfer schema graph has all the information of the original schema graph.)

Figure VI.3 shows the authority transfer schema graph that corresponds to the schema graph of Figure VI.2 (the edge labels are omitted). The motivation for defining two edges for each edge of the schema graph is that authority potentially flows in both directions and not only in the direction that appears in the schema. For example, a paper passes its authority to its authors and vice versa. Notice however, that the authority flow in each direction (defined by the authority transfer rate) may not be the same. For example, a paper that is cited by important papers is clearly important but citing important papers does not make a paper important.

Notice that the sum of authority transfer rates of the outgoing edges of a schema node u may be less than 1[1], if the administrator believes that the edges starting from u do not transfer much authority. For example, in Figure VI.3, conferences only transfer 30% of their authority.

Authority Transfer Data Graph. Given a data graph $D(V_D, E_D)$ that conforms to an authority transfer schema graph $G^A(V_G, E^A)$, ObjectRank derives an

[1] In terms of the random walk model, this would be equivalent to the disappearance of a surfer.

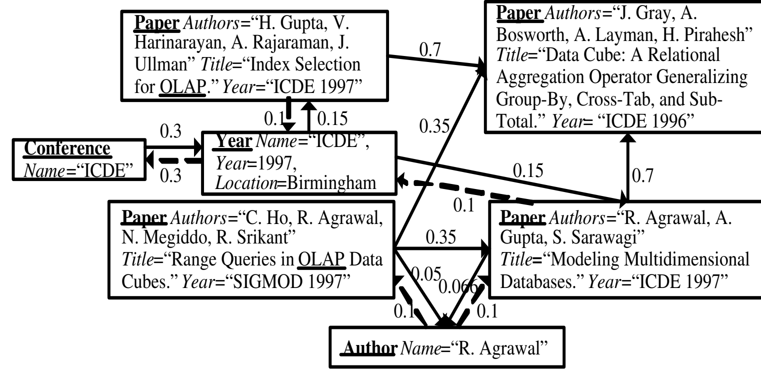


Figure VI.4: Authority transfer data graph

authority transfer data graph $D^A(V_D, E_D^A)$ (Figure VI.4) as follows. For every edge $e = (u \rightarrow v) \in E_D$ the authority transfer data graph has two edges $e^f = (u \rightarrow v)$ and $e^b = (v \rightarrow u)$. The edges e^f and e^b are annotated with authority transfer rates $\alpha(e^f)$ and $\alpha(e^b)$. Assuming that e^f is of type e_G^f, then

$$\alpha(e^f) = \begin{cases} \frac{\alpha(e_G^f)}{OutDeg(u, e_G^f)}, \; if \; OutDeg(u, e_G^f) > 0 \\ 0, \; if \; OutDeg(u, e_G^f) = 0 \end{cases} \tag{VI.4}$$

where $OutDeg(u, e_G^f)$ is the number of outgoing edges from u, of type e_G^f. The authority transfer rate $\alpha(e^b)$ is defined similarly. Figure VI.4 illustrates the authority transfer data graph that corresponds to the data graph of Figure VI.1 and the authority schema transfer graph of Figure VI.3. Notice that the sum of authority transfer rates of the outgoing edges of a node u of type $\mu(u)$ may be less than the sum of authority transfer rates of the outgoing edges of $\mu(u)$ in the authority transfer schema graph, if u does not have all types of outgoing edges.

VI.B.2 Importance vs. Relevance.

The authority flow factor described in this chapter can be applied to all the criteria of Section II.C. However, as we mentioned above we do not tackle the "association between nodes of result-tree" criterion. The other two criteria

("global importance" and "relevance to keywords" are combined and calculated as we describe below.

The score of a node v with respect to a keyword query w is a combination of the global ObjectRank $r^G(v)$ of v and the keyword-specific ObjectRank $r^w(v)$. We propose the following combining function, although other functions may be used as well:

$$r^{w,G}(v) = r^w(v) \cdot (r^G(v))^g \tag{VI.5}$$

where g is the *global ObjectRank weight*, which determines how important the global ObjectRank is. Notice that g may be accessible to the users or fixed by the administrator. The calculations of the keyword-specific and the global ObjectRank are performed as follows (we assume single-keyword queries at this point).

Keyword-specific ObjectRank. Given a single keyword query w, ObjectRank finds the *keyword base set* $S(w)$ (from now on referred to simply as base set when the keyword is implied) of objects that contain the keyword w and assigns an ObjectRank $r^w(v_i)$ to every node $v_i \in V_D$ by resolving the equation

$$\mathbf{r}^w = d\mathbf{A}\mathbf{r}^w + \frac{(1-d)}{|S(w)|}\mathbf{s} \tag{VI.6}$$

where $A_{ij} = \alpha(e)$ if there is an edge $e = (v_j \rightarrow v_i)$ in E_D^A and 0 otherwise, d controls the base set importance, and $\mathbf{s} = [s_1, \ldots, s_n]^T$ is the base set vector for $S(w)$, i.e., $s_i = 1$ if $v_i \in S(w)$ and $s_i = 0$ otherwise.

The damping factor d determines the portion of ObjectRank that an object transfers to its neighbors as opposed to keeping to itself. It was first introduced in the original PageRank paper [11], where it was used to ensure convergence in the case of PageRank sinks. However, in addition to that, in our work it is a calibrating factor, since by decreasing d, we favor objects that actually contain the keywords (i.e., are in base set)as opposed to objects that acquire ObjectRank through the incoming edges. Typical values for d are 0.85 for normal behavior and

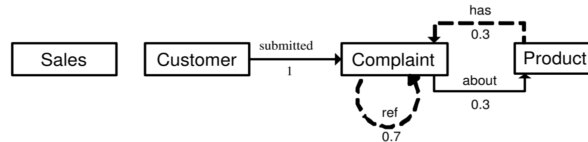


Figure VI.5: Authority transfer schema graph for Complaints database.

0.1 to favor objects that actually contain the keywords.

Global ObjectRank. The definition of global ObjectRank is different for different applications or even users of the same application. In this chapter, we focus on cases where the global ObjectRank is calculated applying the random surfer model, and including all nodes in the base set. The same calibrating parameters are available, as in the keyword-specific ObjectRank. Notice that this way of calculating the global ObjectRank, which is similar to the PageRank approach [11], assumes that all nodes (pages in PageRank) initially have the same value. However, there are many applications where this is not true.

For example, consider a complaints database (Figure VI.5), which stores the complaint reports of customers regarding products of the company. Also, each complaint may reference another complaint, and there is a sales table. Assume we wish to rank the complaint reports according to their urgency, given that the goal of the company is to keep the "good" customers satisfied, and the "goodness" of a customer is the total sales associated with him/her. Then, the base set for the computation of the global ObjectRank is the set of customers, and each customer is given a base ObjectRank proportional to his/her total sales amount. A reasonable assignment of authority transfer rates is shown in Figure VI.5.

VI.B.3 Multiple-Keywords Queries.

We define the semantics of a multiple-keyword query "$w_1, \ldots, w_m$" by naturally extending the random walk model. We consider m independent random surfers, where the ith surfer starts from the keyword base set $S(w_i)$. For AND

(a)

47.31	11.44	An XML Indexing Structure with Relative Region Coordinate. Dao Dinh Kha, ICDE 2001
41.02	3.08	DataGuides: Enabling Query ... Optimization in Semistructured... Roy Goldman, VLDB 1997
7.44	28.43	Access Path Selection in a RDBMS. Patricia G. Selinger, SIGMOD 1979
31.44	3.24	Querying Object-Oriented Databases. Michael Kifer, SIGMOD 1992
26.73	3.09	A Query … Optimization Techniques for Unstructured Data. Peter Buneman, SIGMOD 1996

(b)

47.31	11.44	An XML Indexing Structure with Relative Region Coordinate. Dao Dinh Kha, ICDE 2001
7.44	28.43	Access Path Selection in a RDBMS. Patricia G. Selinger, SIGMOD 1979
2.04	102.1	R-Trees: A Dynamic Index Structure for Spatial Searching. Antonin Guttman, SIGMOD 1984
1.73	112.7	The K-D-B-Tree: A Search Structure For Large … Indexes. John T. Robinson, SIGMOD 1981
41.02	3.08	DataGuides: Enabling Query … Optimization in Semistructured... Roy Goldman, VLDB 1997

Figure VI.6: Top 5 papers on “XML Index”, with and without emphasis on “XML”

semantics, the ObjectRank of an object v with respect to the m-keywords query is the probability that, at a given point in time, the m random surfers are simultaneously at v. Hence the ObjectRank $r^{w_1,\ldots,w_m}(v)$ of the node v with respect to the m keywords is

$$r^{w_1,\ldots,w_m}(v) = \prod_{i=1,\ldots,m} r^{w_i}(v) \qquad \text{(VI.7)}$$

where $r^{w_i}(v)$ is the ObjectRank with respect to the keyword w_i.

For OR semantics, the ObjectRank of v is the probability that, at a given point in time, *at least one* of the m random surfers will reach v. Hence, for two keywords w_1 and w_2 it is

$$r^{w_1,w_2}(v) = r^{w_1}(v) + r^{w_2}(v) - r^{w_1}(v)r^{w_2}(v) \qquad \text{(VI.8)}$$

and for more than two, it is defined accordingly. Notice that [29] also sums the topic-sensitive PageRanks to calculate the PageRank of a page.

VI.B.4 Weigh keywords by frequency

A drawback of the *combining function* of Equation VI.7 is that it favors the more popular keywords in the query. The reason is that the distribution of ObjectRank values is more skewed when the size $|S(w)|$ of the base set $S(w)$ increases, because the top objects tend to receive more references. For example, consider two results for the query “XML AND Index” shown in Figure VI.6. Result (b) corresponds to the model described above. It noticeably favors the “Index” keyword

over the “XML”. The first paper is the only one in the database that contains both keywords in the title. However, the next three results are all classic works on indexing and do not apply directly to XML. Intuitively, “XML” as a more specific keyword is more important to the user. Indeed, the result of Figure VI.6 (a) was overwhelmingly preferred over the result of Figure VI.6 (b) by participants of our relevance feedback survey (Section VI.E.1). The latter result contains important works on indexing in semistructured, unstructured, and object-oriented databases, which are more relevant to indexing of XML data. This result is obtained by using the modified formula:

$$r^{w_1,\dots,w_m}(v) = \prod_{i=1,\dots,m} (r^{w_i}(v))^{g(w_i)} \tag{VI.9}$$

where $g(w_i)$ is a *normalizing exponent*, set to $g(w_i) = 1/log(|S(w_i)|)$. Using the normalizing exponents $g(\text{“XML”})$ and $g(\text{“Index”})$ in the above example is equivalent to running in parallel $g(\text{“XML”})$ and $g(\text{“Index”})$ random walks for the “XML” and the “Index” keywords respectively.

VI.B.5 Compare to single base set approach

One can imagine alternative semantics to calculate the ObjectRank for multiple keywords, other than combining the single-keyword ObjectRanks. In particular, consider combining all objects with at least one of the keywords into a single base set. Then a single execution of the ObjectRank algorithm is used to determine the scores of the objects. Incidentally, these semantics were used in the HITS system [38]. We show that such “single base set” semantics can be achieved by combining single-keyword ObjectRank values applying appropriate exponents. Furthermore, we explain how our semantics avoid certain problems of “single base set” semantics.

In order to compare to the “single base set” approach for AND semantics (Equation VI.7), we consider two scenarios and assume without loss of generality that there are two keywords. First, assume that we only put in the base set

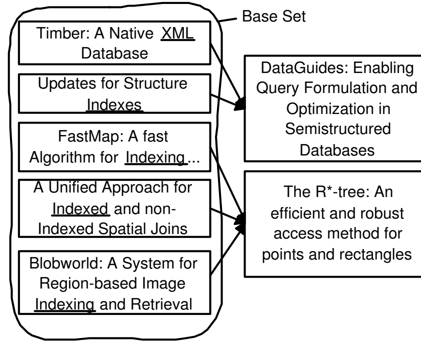


Figure VI.7: Example where "HITS" approach fails in AND semantics.

S objects that contain both keywords. These objects will be in both keyword-specific base sets as well, so these objects and objects pointed by them will receive a top rank in both approaches. Second, if S contains objects containing any of the two keywords, we may end up ranking highest an object that is only pointed by objects containing one keyword. This cannot happen with the keyword-specific base sets approach. For example, in Figure VI.7, the "single base set" approach would rank the R^* paper higher than the DataGuides paper for the query "XML AND Index", even though the R^* paper is irrelevant to XML.

For OR semantics (Equation VI.8), the base set S in the"single base set" approach is the union of the keyword-specific base sets. We compare to an improved version of the "single base set" approach, where objects in base set are weighted according to the keywords they contain, such that infrequent keywords are assigned higher weight. In particular, if an object contains both keywords, for a two keyword query, it is assigned a base ObjectRank of $(1-d) \cdot (\frac{1}{|S(w_1)|} + \frac{1}{|S(w_2)|})$. Then, using the Linearity Theorem in [36], we can prove that the ObjectRanks calculated by both approaches are the same.

VI.C Architecture

Figure VI.8 shows the architecture of the ObjectRank system, which is divided into two stages. The preprocessing stage consists of the *ObjectRank Execu-*

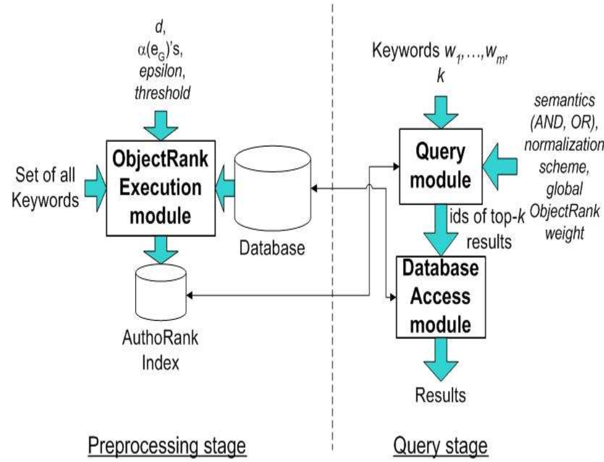


Figure VI.8: System Architecture.

tion module, which inputs the database to be indexed, the set of all keywords that will be indexed, and a set of parameters (the rest of the adjusting parameters are input during the query stage). In particular these parameters are: (i) the damping factor d, (ii) the authority transfer rates $\alpha(e_G)$'s of the authority transfer schema graph G^A, (iii) the convergence constant *epsilon* which determines when the ObjectRank algorithm converges, and (iv) the *threshold* value which determines the minimum ObjectRank that an object must have to be stored in the ObjectRank Index.

The ObjectRank Execution module creates the *ObjectRank Index*, which is an inverted index, indexed by the keywords. For each keyword w, it stores a list of $\langle id(u), r^w(u)\rangle$ pairs for each object u that has $r^w(u) \geq threshold$. The pairs are sorted by descending $r^w(u)$ to facilitate an efficient querying method as we describe below. The ObjectRank Index has been implemented as an index-based table, where the lists are stored in a CLOB attribute. A hash-index is built on top of each list to allow for random access, which is required by the Query module.

The *Query module* inputs a set of sorted $\langle id(u), r^w(u)\rangle$ pairs lists $L_1, \ldots, L_m$ and a set of adjusting parameters, and outputs the top-k objects according to the combining function (Equation VI.7 or VI.8). In particular, these parameters are: (i) the semantics to be used (AND or OR), (ii) the normalization

scheme, i.e., the exponents to use, and (iii) the global ObjectRank weight. The naive approach would be to make one pass of all lists to calculate the final ObjectRank values for each object and then sort this list by final ObjectRank. Instead, we use the *Threshold Algorithm* [17] which is guaranteed to read the minimum prefix of each list. Notice that the Threshold Algorithm is applicable since both combining functions (Equations VI.7 and VI.8) are monotone, and random access is possible on the stored lists.

Finally, the *Database Access module* inputs the result *id*s and queries the database to get the suitable information to present the objects to the user. This information is stored into an id-indexed table, that contains a CLOB attribute value for each object id. For example, a paper object CLOB would contain the paper title, the authors' names, and the conference name and year.

VI.D Index Creation Algorithms

This section presents algorithms to create the ObjectRank index. Section VI.D.1 presents an algorithm for the case of arbitrary authority transfer data graphs D^A. Sections VI.D.2 and VI.D.3 show how we can do better when D^A is a directed acyclic graph (DAG) and "almost" a DAG respectively (the latter property is explained in Section VI.D.3). In Sections VI.D.4 and VI.D.5, we present optimizations when the authority transfer graph has a small vertex cover, or is a DAG of subgraphs. Finally, Section VI.D.6 presents optimization opportunities based on manipulating the initial values of the iterative algorithm.

VI.D.1 General algorithm

Figure VI.9 shows the algorithm that creates the ObjectRank Index. The algorithm accesses the authority transfer data graph D^A many times, which may lead to a too long execution time if D^A is very large. Notice that this is usually not a problem, since D^A only stores object ids and a set of edges which is small enough

```
CreateIndex(keywordsList, epsilon, threshold, α(.), d){
01. For each keyword w in keywordsList do {
02.   While not converged do
03.   /*i.e., ∃v, |r^(k+1)(v) − r^(k)(v)| > epsilon*/
04.     MakeOnePass(w,α(.), d);
05.   StoreObjectRanks();
06. }
}
MakeOnePass(w,α(.), d) {
07. Evaluate Equation VI.6 using the r from the previous iteration on the right side;
}
StoreObjectRanks() {
08. Sort the ⟨id(i), r(v_i)⟩ pairs list by r(v_i) and store it in inverted index,
   after removing pairs with r(v_i) < threshold;
}
```

Figure VI.9: Algorithm to create ObjectRank Index

to fit into main memory for most databases. Notice that lines 2-4 correspond to the original PageRank calculation [11] modulo the authority transfer rates information.

VI.D.2 DAG algorithm

There are many applications where the authority transfer data graph is a DAG. For example a database of papers and their citations (ignoring author and conference objects), where each paper only cites previously published papers, is a DAG. Figure VI.10 shows an improved algorithm, which makes a single pass of the graph D^A and computes the actual ObjectRank values. Notice that there is no need for *epsilon* any more since we derive the precise solution of Equation VI.6, in contrast to the algorithm of Figure VI.9 which calculates approximate values. The intuition is that ObjectRank is only transferred in the direction of the topological ordering, so a single pass suffices. Notice that topologically sorting a graph $G(V, E)$ takes time $\Theta(V + E)$ [15] in the general case. In many cases the semantics of the

database can lead to a better algorithm. For example, in the papers database, we can efficiently topologically sort the papers by first sorting the conferences by date. This method is applicable for databases where a temporal or other kind of ordering is implied by the link structure.

```
CreateIndexDAG(keywordsList, threshold, α(.), d){
01. Topologically sort nodes in graph D^A;
02. /*Consecutive accesses to D'^A are in topological order.*/
03. For each keyword w in keywordsList do {
04.   MakeOnePass(w,α(.), d);
05.   StoreObjectRanks();
06. }
}
```

Figure VI.10: Algorithm to create ObjectRank Index for DAGs

In the above example, the DAG property was implied by the semantics. However, in some cases we can infer this property by the structure of the authority transfer schema graph G^A, as the following theorem shows.

Theorem 10 *The authority transfer data graph D^A is a DAG if and only if*

- *the authority transfer schema graph G^A is a DAG, or*
- *for every cycle c in G^A, the subgraph D'^A of D^A consisting of the nodes (and the edges connecting them), whose type is one of the schema nodes of c, is a DAG.*

VI.D.3 Almost-DAG algorithm

The most practically interesting case is when the authority transfer data graph D^A is *almost* a DAG, that is, there is a "small" set U of *backedges*, and if these edges are removed, D^A becomes a DAG. Notice that the set U is not unique, that is, there can be many *minimal* (i.e., no edge can be removed from U) sets of

backedges. Instead of working with the set of backedges U, we work with the set L of *backnodes*, that is, nodes from which the backedges start. This reduces the number of needed variables as we show below, since $|L| \leq |U|$.

In the papers database example (when author and conference objects are ignored), L is the set of papers citing a paper that was not published previously. Similarly, in the complaints database (Figure VI.5), most complaints reference previous complaints. Identifying the minimum set of backnodes is NP-complete[2] in the general case. However, the semantics of the database can lead to efficient algorithms. For example, for the databases we discuss in this paper (i.e, the papers and the complaints databases), a backnode is simply an object referencing an object with a newer timestamp.

The intuition of the algorithm (Figure VI.11) is as follows: the ObjectRank of each node can be split to the DAG-ObjectRank which is calculated ignoring the backedges, and the backedges-ObjectRank which is due to the backedges.

To calculate backedges-ObjectRank we assign a variable c_i to each backnode c_i (for brevity, we use the same symbol to denote a backnode and its ObjectRank), denoting its ObjectRank. Before doing any keyword-specific calculation, we calculate how c_i's are propagated to the rest of the graph D^A (line 5), and store this information in $\mathbf{C}$. Hence C_{ij} is the coefficient with which to multiply c_j when calculating the ObjectRank of node v_i. To calculate $\mathbf{C}$ (lines 13-15) we assume that the backedges are the only source of ObjectRank, and make one pass of the DAG in topological order.

Then, for each keyword-specific base set: (a) we calculate the DAG-ObjectRanks $\mathbf{r}'$ (line 7) ignoring the backedges (but taking them into account when calculating the outgoing degrees), (b) calculate c_i's solving a linear system (line 8), and (c) calculate the total ObjectRanks (line 10) by adding the backedge-ObjectRank ($\mathbf{C} \cdot \mathbf{c}$) and the DAG-ObjectRank($\mathbf{r}'$). Each line of the system of line 8 corresponds to a backnode $c_i \equiv v_j$ (i.e., the ith backnode is the jth node of the

[2]Proven by reducing Vertex Cover to it.

```
CreateIndexAlmostDAG(keywordsList, threshold, α(.), d){
01. c: vector of ObjectRanks of backnodes;
02. Identify backnodes, and topologically sort the DAG (D^A without the backedges) D'^A;
03. /*Consecutive accesses to D'^A are in topological order.*/
04. /*Backedges are considered in D'^A for α(.) .*/
05. C=BuildCoefficientsTable();
06. For each keyword w in keywordsList do {
07.   Calculate ObjectRanks vector r' for D'^A executing MakeOnePass(w,α(.), d);
08.   Solve c = C̄ · c + r̄';
09.   /*D̄ denotes keeping only the lines of D corresponding to backnodes.*/
10.   r = C · c + r'
11.   StoreObjectRanks();
12.   }
}
BuildCoefficientsTable(){
13. For each node v_j do
14.   r(v_j) = d · Σ_{backnode c_i points at v_j} (α(c_i → v_j) · c_i)+
              d · Σ_{non-backnode v_l points at v_j} (α(v_l → v_j) · r(v_l));
15. Return C, such that r = C · c
}
```

Figure VI.11: Algorithm to create ObjectRank Index for *almost* DAGs

topologically sorted authority transfer data graph D'^A), whose ObjectRank c_i is the sum of the backedge-ObjectRank ($\mathbf{C}_j \cdot \mathbf{c}$) and the DAG-ObjectRank ($\mathbf{r}'_j$). The overline notation on the matrices of this equation selects the L lines from each table that correspond to the backnodes. We further explain the algorithm using an example.

Example 6 *The graph D^A is shown in Figure VI.12 (a). Assume $d = 0.5$ and all edges are of the same type t with authority transfer rate $\alpha(t) = 1$. First we topologically sort the graph and identify the backnodes $c_1 \equiv P_5, c_2 \equiv P_4$. Then we create the coefficients table C (line 5), as follows:*

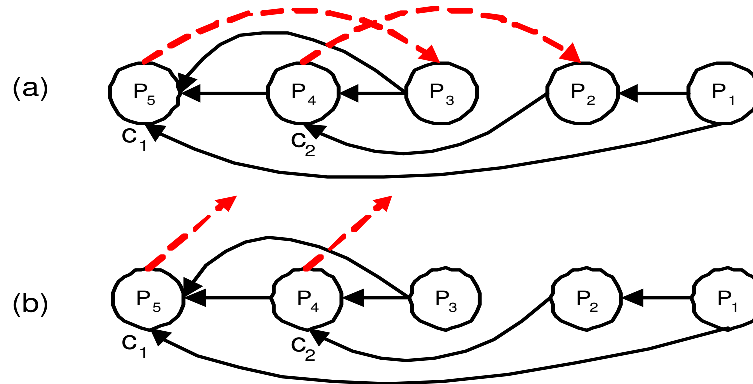


Figure VI.12: Almost DAG.

$$
\begin{aligned}
r(P_1) &= 0 \\
r(P_2) &= 0.5 \cdot 0.5 \cdot c_2 = 0.25 \cdot c_2 \\
r(P_3) &= 0.5 \cdot c_1 \\
r(P_4) &= 0.5 \cdot r(P_2) + 0.5 \cdot 0.5 \cdot r(P_3) = 0.125 \cdot c_1 + 0.125 \cdot c_2 \\
r(P_5) &= 0.5 \cdot 0.5 \cdot r(P_3) + 0.5 \cdot 0.5 \cdot r(P_4) = 0.156 \cdot c_1 + 0.031 \cdot c_2
\end{aligned}
$$

$$
\mathbf{C} = \begin{bmatrix} 0 & 0 \\ 0 & 0.25 \\ 0.5 & 0 \\ 0.125 & 0.125 \\ 0.156 & 0.031 \end{bmatrix}
$$

Assume we build the index for one keyword w contained in nodes P_1, P_3. We calculate (line 7) ObjectRanks for D'^A (taken by removing the backedges (dotted lines) from D^A).

$$
\begin{aligned}
r(P_1) &= 0.5 \\
r(P_2) &= 0.5 \cdot 0.5 \cdot r(P_1) = 0.125 \\
r(P_3) &= 0.5 \\
r(P_4) &= 0.5 \cdot 0.5 \cdot r(P_3) + 0.5 \cdot r(P_2) = 0.188 \\
r(P_5) &= 0.5 \cdot 0.5 \cdot r(P_4) + 0.5 \cdot 0.5 \cdot r(P_3) + 0.5 \cdot 0.5 \cdot r(P_1) = 0.297
\end{aligned}
$$

$\mathbf{r}' = [0.5\ 0.125\ 0.5\ 0.188\ 0.297]^T$

Solving the equation of line 8:

$$
\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0.156 & 0.031 \\ 0.125 & 0.125 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + \begin{bmatrix} 0.297 \\ 0.188 \end{bmatrix}
$$

we get: $\mathbf{c} = [0.361\ 0.263]^T$, *where the overline-notation selects from the matrices the 5-th and the 4-th lines, which correspond to the backnodes* c_1 *and* c_2 *respectively. The final ObjectRanks are (line 10):* $\mathbf{r} = [0.5\ 0.190\ 0.680\ 0.266\ 0.361]^T$.

This algorithm can be viewed as a way to reduce the $n \times n$ ObjectRank calculation system of Equation VI.6, where n is the size of the graph, to the much smaller $|L| \times |L|$ equations system of line 8 of Figure VI.11. Interestingly, the two equations systems have the same format $\mathbf{r} = \mathbf{A}\mathbf{r} + \mathbf{b}$, only with different coefficient tables $\mathbf{A}, \mathbf{b}$. The degree of reduction achieved is inversely proportional to the number of backnodes.

The linear, first-degree equations system of line 8 can be solved using any of the well-studied arithmetic methods like Jacobi and Gauss-Seidel [23], or even using the PageRank iterative approach which is simpler because we do not have to solve each equation with respect to a variable. The latter is shown to perform better in Section VI.E.2.

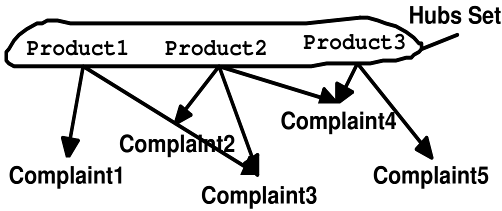


Figure VI.13: Hierarchical-graph.

VI.D.4 Algorithm for graphs with small vertex cover

Similarly to the almost-DAG case, we can reduce the ObjectRank calculation to a much smaller system (than the one of Equation VI.6) if authority transfer data graph D^A contains a relatively small vertex cover H. For example, consider a subset of the complaints database (Figure VI.5) consisting of the products and the complaints (without the reference edge to other complaints). Then H is the set of the products (Figure VI.13).[3] We call the nodes of H hub-nodes.

The intuition of the algorithm is the following: Let $r(h_i)$ be the ObjectRank of hub-node h_i. First, the ObjectRank of every non-hub-node i is expressed as a function of the ObjectRanks of the hub-nodes pointing to i. Then the $r(h_i)$ is expressed as a function of the non-hub-nodes pointing to h_i. This expression is equal to $r(h_i)$, so we get $|H|$ such equations for the $|H|$ hub-nodes. Hence we reduce the computation to a $|H| \times |H|$ linear, first-degree system. Notice that we omit the details of the optimizations of Sections VI.D.4 and VI.D.5 due to lack of space.

VI.D.5 Serializing ObjectRank Calculation

This section shows when and how we can *serialize* the ObjectRank calculation of the whole graph $D^A(V_D, E_D^A)$ over ObjectRank calculations for disjoint, non-empty subsets $L_1, \ldots, L_r$ of V_D, where $L_1 \cup \cdots \cup L_r \equiv V_D$. The calculation is serializable if we first calculate the ObjectRanks for L_1, then use these ObjectRanks to calculate the ObjectRanks of L_2 and so on.

[3]A complaint can refer to more than one products.

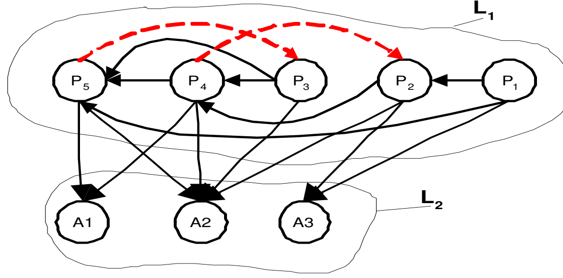


Figure VI.14: Serializable Graph.

For example, consider the subset of the papers database consisting of the papers, their citations and the authors, where authority is transferred between the papers and from a paper to its authors (and not vice versa). Figure VI.14 shows how this authority transfer data graph can be serialized. In particular, we first calculate the ObjectRanks for the nodes in L_1 and then for the nodes in L_2, as we elaborate below.

To define when the calculation is serializable, we first define the graph $D'^A(V', E')$ with $V' = \{L_1 \cup \cdots \cup L_r\}$ and $E' = \{(L_i, L_j) | \exists (v_i, v_j) \in E^A_D \wedge v_i \in L_i \wedge v_j \in L_j\}$. That is, there is an edge (L_i, L_j) in D'^A if there is an edge between two nodes $v_i \in L_i, v_j \in L_j$ of D^A. The following theorem defines when the ObjectRank calculation is serializable.

Theorem 11 *The ObjectRank calculation for D^A is serializable iff D'^A is a DAG.*

The algorithm works as follows: Let $L_1, \ldots, L_r$ be topologically ordered. First, the ObjectRanks of the nodes in L_1 are computed ignoring the rest of D^A. Then we do the same for L_2, including in the computation the set I of nodes (and the corresponding connecting edges) of L_1 connected to nodes in L_2. Notice that the ObjectRanks of the nodes in I are not changed since there is no incoming edge from any node of L_2 to any node in I. Notice that any of the ObjectRank calculations methods described above can be used in each subset L_i.

VI.D.6 Manipulating Initial ObjectRank values

All algorithms so far assume that we do a fresh execution of the algorithm for every keyword. However, intuitively we expect nodes with high global ObjectRank to also have high ObjectRank with respect to many keywords. We exploit this observation by assigning the global ObjectRanks as initial values for each keyword specific calculation.

Furthermore, we investigate a space vs. time tradeoff. In particular, assume we have limitations on the index size. Then we only store a prefix (the first p nodes) of the nodes' list (recall that the lists are ordered by ObjectRank) for each keyword. During the query stage, we use these values as initial values for the p nodes and a constant (we experimentally found 0.03 to be the most efficient for our datasets) for the rest[4]. Both ideas are experimentally evaluated in Section VI.E.2.

VI.E Experiments

In Section VI.E.1 we show that the ObjectRank system produces results of high quality. Then in Section VI.E.2 we evaluate the performance of the system.

VI.E.1 Quality Evaluation

To evaluate the quality of the results of ObjectRank, we conducted two surveys. The first was performed on the DBLP database, with eight professors and Ph.D. students from the UC, San Diego database lab, who were not involved with the project. The second survey used the publications database of the IEEE Communications Society (COMSOC) [5] and involved five senior Ph.D. students from the Electrical Engineering Department.

[4]Notice that, as we experimentally found, using the global ObjectRanks instead of a constant for the rest nodes is less efficient. The reason is that if a node u is not in the top-p nodes for keyword k, u probably has a very small ObjectRank with respect to k. However u may have a great global ObjectRank.

[5]http://www.comsoc.org

Each participant was asked to compare and rank two to five lists of top-10 results for a set of keyword queries, assigning a score of 1 to 10, according to the relevance of the results list to the query. Each result list was generated by a different variation of the ObjectRank algorithm. One of the results lists in each set was generated by the "default" ObjectRank configuration which used the authority transfer schema graph of Figure VI.3 and $d = 0.85$. The users knew nothing about the algorithms that produced each result list. The survey was designed to investigate the quality of ObjectRank when compared to other approaches or when changing the adjusting parameters.

Effect of keyword-specific ranking. First, we assess the basic principle of ObjectRank, which is the keyword-specific scores. In particular, we compared the default (that is, with the parameters set to the values discussed in Section VI.B) ObjectRank with the global ranking algorithm that sorts objects that contain the keywords according to their global ObjectRank (where the base-set contains all nodes). Notice that this is equivalent to what Google does for Web pages, modulo some minor difference on the calculation of the relevance score by Google. The DBLP survey included results for two keyword queries: "OLAP" and "XML". The score was 7:1 and 5:3 in favor of the keyword-specific ObjectRank for the first and second keyword query respectively. The COMSOC survey used the keywords "CDMA" and "UWB (ultra wideband)" and the scores were 4:1 and 5:0 in favor of the keyword-specific approach respectively.

Effect of authority transfer rates. We compared results of the default ObjectRank with a simpler version of the algorithm that did not use different authority transfer rates for different edge types, i.e., all edge types were treated equally. In the DBLP survey, for both keyword queries, "OLAP" and "XML", the default ObjectRank won with scores 5:3 and 6.5:1.5 (the half point means that a user thought that both rankings were equally good) respectively. In the COMSOC survey, the scores for "CDMA" and "UWB" were 3.5:1.5 and 5:0 respectively.

Effect of the damping factor d. We tested three different values of the damping factor d: 0.1, 0.85, and 0.99, for the keyword queries "XML" and "XML AND Index" on the DBLP dataset. Two points were given to the first choice of a user and one point to the second. The scores were $2.5 : 8 : 13.5$ and $10.5 : 11.5 : 2$ (the sum is 24 since there are 8 users times 3 points per query) respectively for the three d values. We see that higher d values are preferred for the "XML", because "XML" is a very large area. In contrast, small d are preferable for "XML AND Index", because few papers are closely related to both keywords, and these papers typically contain both of them. The results were also mixed in the COMSOC survey. In particular, the damping factors 0.1, 0.85, and 0.99 received scores of 5:6:4 and 4.5:3.5:7 for the queries "CDMA" and "UWB" respectively.

Effect of changing the weights of the keywords. We compared the combining functions for AND semantics of Equations VI.7 and VI.9 for the two-keyword queries "XML AND Index" and "XML AND Query", in the DBLP survey. The use of the normalizing exponents proposed in Section VI.B.3 was preferred over the simple product function with ratios of 6:2 and 6.5:1.5 respectively. In the COMSOC survey, the same experiment was repeated for the keyword query "diversity combining". The use of normalizing exponents was preferred at a ratio of 3.5:1.5.

VI.E.2 Performance Experiments

In this section we experimentally evaluate the system and show that calculating the ObjectRank is feasible, both in the preprocessing and in the query execution stage. For the evaluation we use two real and a set of synthetic datasets: COMSOC is the dataset of the publications of the IEEE Communications Society [6], which consists of $55,000$ nodes and $165,000$ edges. DBLPreal is a subset of the DBLP dataset, consisting of the publications in twelve database conferences. This dataset contains $13,700$ nodes and $101,500$ edges. However, these datasets are too small to evaluate the index creation algorithms. Hence, we also created a set of

[6] http://www.comsoc.org

artificial datasets shown in Table VI.2, using the words of the DBLP dataset. The outgoing edges are distributed uniformly among papers, that is, each paper cites on average 10 other papers. The incoming edges are assigned by a non-uniform random function, similar to the one used in the TPC-C benchmark [7], such that the top-10% of the most cited papers receive 70% of all the citations.

name	*#nodes*	*#edges*
DBLP30	3,000	30,000
DBLP100	10,000	100,000
DBLP300	30,000	300,000
DBLP1000	100,000	1,000,000
DBLP3000	300,000	3,000,000

Table VI.2: Synthetic Datasets.

To store the databases in a RDBMS, we decomposed them into relations according to the relational schema shown in Figure VI.15. 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*. We ran our experiments using the Oracle 9i RDBMS on a Xeon 2.2-GHz PC with 1 GB of RAM. We implemented the preprocessing and query-processing algorithms in Java, and connect to the RDBMS through JDBC.

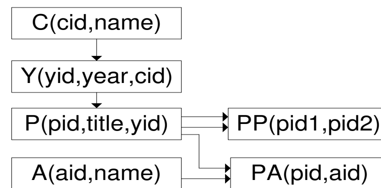


Figure VI.15: Relational schema.

The experiments are divided into two classes. First, we measure how fast the ObjectRank Execution module (Figure I.7) calculates the ObjectRanks for all keywords and stores them into the ObjectRank Index, using the *CreateIndex*

[7] http://www.tpc.org/tpcc/

algorithm of Figure VI.9. The size of the ObjectRank Index is also measured. This experiment is repeated for various values of *epsilon* and *threshold*, and various dataset sizes. Furthermore, the General ObjectRank algorithm is compared to the almost-DAG algorithm, and the effect of using various initial ObjectRank values is evaluated. Second, the Query module (Figure VI.8) is evaluated. In particular, we measure the execution times of the combining algorithm (Section VI.C) to produce the top-k results, for various values of k, various numbers of keywords m, and OR and AND semantics.

Preprocessing stage

threshold	*time (sec)*	*nodes/keyword*	*size (MB)*
0.3	3702	84	2.20
0.5	3702	67	1.77
1.0	3702	46	1.26

Table VI.3: Index Creation for DBLPreal for *epsilon* $= 0.1$

threshold	*time (sec)*	*nodes/keyword*	*size (MB)*
0.05	80829	9.4	1.17
0.07	80829	8.3	1.08
0.1	80829	7.7	1.03

Table VI.4: Index Creation for COMSOC for *epsilon* $= 0.05$

epsilon	*time (sec)*	*nodes/keyword*	*size (MB)*
0.05	3875	67	1.77
0.1	3702	67	1.77
0.3	3517	67	1.77

Table VI.5: Index Creation for DBLPreal for *threshold* $= 0.5$

General ObjectRank algorithm. Tables VI.3 and VI.4 show how the storage space for the ObjectRank index decreases as the ObjectRank *threshold* of the stored objects increases, for the real datasets. Notice that DBLPreal and COMSOC have $12,341$ and $40,577$ keywords respectively. Also notice that much fewer nodes per

epsilon	*time (sec)*	*nodes/keyword*	*size (MB)*
0.05	80829	7.7	1.03
0.07	77056	7.7	1.03
0.1	74337	7.7	1.03

Table VI.6: Index Creation for COMSOC for $threshold = 0.1$

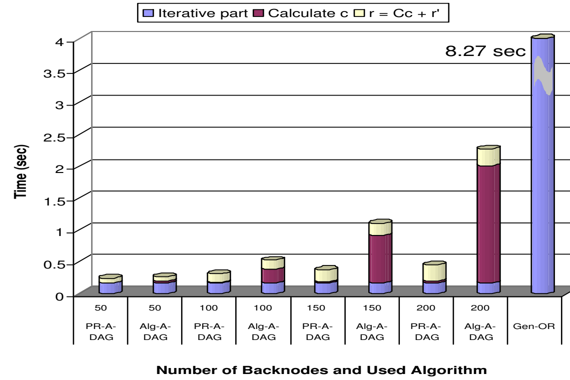


Figure VI.16: Evaluate almost-DAG algorithm.

keyword have ObjectRank above the *threshold* in COMSOC, since this dataset is more sparse and has more keywords. The time to create the index does not change with *threshold* since *threshold* is not used during the main execution loop of the CreateIndex algorithm. Tables VI.5 and VI.6 show how the index build time decreases as *epsilon* increases. The reason is that fewer iterations are needed for the algorithm to converge, on the cost of lower accuracy of the calculated ObjectRanks. Notice that the storage space does not change with *epsilon*, as long as $epsilon < threshold$.

Table VI.7 shows how the execution times and the storage requirements for the ObjectRank index scale with the database size for the DBLP synthetic datasets for $epsilon = 0.05$ and $threshold = 0.1$. Notice that the average number of nodes having ObjectRank higher than the *threshold* increases considerably with the dataset size, because the same keywords appear multiple times.

General ObjectRank vs. almost-DAG algorithm. Figure VI.16 compares the index creation time of the General ObjectRank algorithm (*Gen-OR*) and two

dataset	*time (sec)*	*nodes/keyword*	*size (MB)*
DBLP30	2933	6	0.3
DBLP100	11513	21	0.7
DBLP300	45764	65	1.7
DBLP1000	206034	316	7.9
DBLP3000	6398043	1763	43.6

Table VI.7: Index Creation for Synthetic Datasets.

versions of the almost-DAG algorithm, on the DBLP1000 dataset, for various number of backnodes. The *algebraic* version (*Alg-A-DAG*) precisely solves the $\mathbf{c} = \overline{\mathbf{C}} \cdot \mathbf{c} + \overline{\mathbf{r'}}$ system using an off the self algebraic solver. The *PageRank* version (*PR-A-DAG*) solves this system using the PageRank [11] iterative method. The measured times are the average processing time for a single keyword and do not include the time to retrieve the base-set from the inverted text index, which is common to all methods. Also, the time to calculate $\mathbf{C}$ is omitted, since it $\mathbf{C}$ is calculated once for all keywords, and it requires a single pass over the graph. The *Iterative part* of the execution times corresponds to the one pass we perform on the DAG subgraph to calculate $\mathbf{r'}$ for almost-DAG algorithms, and to the multiple passes which consist the whole computation for the General ObjectRank algorithm.

Also, notice that *epsilon* $= 0.1$ for this experiment (the *threshold* value is irrelevant since it does not affect the processing time, but only the storage space). The time to do the topological sorting is about 20 sec which is negligible compared to the time to calculate the ObjectRanks for all keywords.

Initial ObjectRanks. This experiment shows how the convergence of the General ObjectRank algorithm is accelerated when various values are set as initial ObjectRanks. In particular, we compare the naive approach, where we assign an equal initial ObjectRank to all nodes, to the global-as-initial approach, where the global ObjectRanks are used as initial values for the keyword-specific ObjectRank calculations. We found that on DBLPreal (COMSOC), for *epsilon* $= 0.1$, the naive and global-as-initial approaches take 16.3 (15.8) and 12.8 (13.7) iterations

respectively.

Furthermore, we evaluate the space vs. time tradeoff described in Section VI.D.6. Tables VI.8 and VI.9 show the average number of iterations for *epsilon* = 0.1 on DBLPreal and COMSOC respectively for various values of the precomputed list length p.

List length p	*iterations*
13700	1
13000	1.2
8000	1.8
2500	3
800	8.7
100	13.3
0	16.3

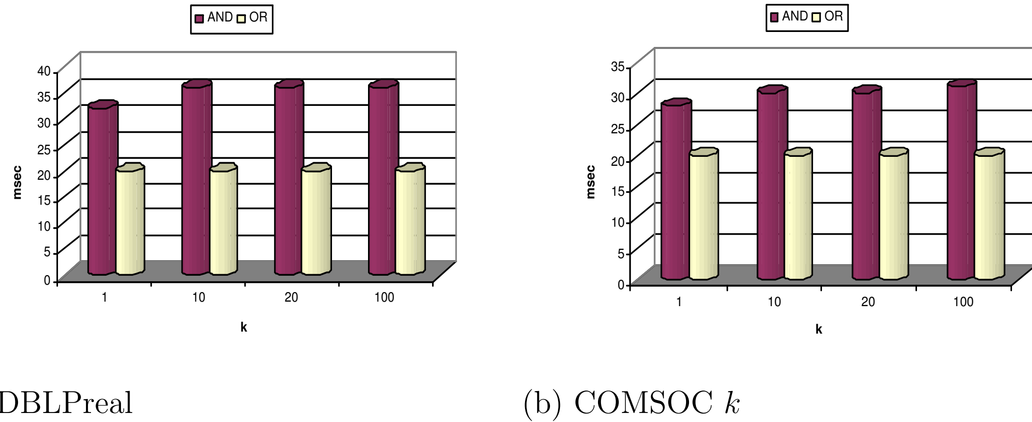
Table VI.8: Number of iterations for various lengths of precomputed lists for DBLPreal

List length p	*iterations*
55000	1
54000	2.9
30000	5.3
13000	6.5
1600	7.8
400	10.7
25	13
0	15.8

Table VI.9: Number of iterations for various lengths of precomputed lists for COMSOC

Query Stage Experiments

Figures VI.17 (a) and VI.17 (b) show how the average execution time changes for varying number of requested results k, for two-keyword queries on DBLPreal and COMSOC respectively. We used the index table created with *epsilon* = 0.1 (0.05) and *threshold* = 0.3 (0.1) for DBLPreal (COMSOC). The



(a) DBLPreal (b) COMSOC k

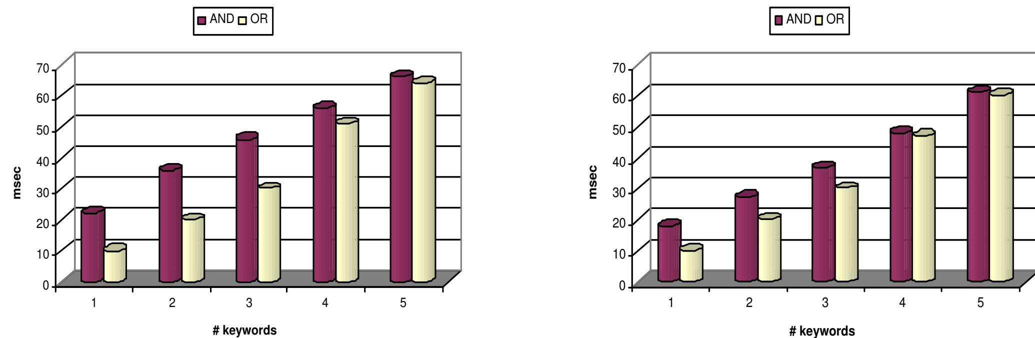
Figure VI.17: Varying k

times are averaged over 100 repetitions of the experiment. Notice that the time does not increase considerably with k, due to the fact that about the same number of random accesses are needed for small k values, and the processing time using the Threshold Algorithm is too small. Also notice that the times for COMSOC are slightly smaller than DBLPreal, because the inverted lists are shorter. Figures VI.18 (a) and VI.18 (b) show that the execution time increases almost linearly with the number of keywords, which again is due to the fact that the disk access time to the ObjectRank lists is the dominant factor, since the processing time is too small. Finally, notice that the execution times are shorter for OR semantics, because there are more results, which leads to a smaller prefix of the lists being read, in order to get the top-k results.

VI.F Related Work

We first present how state-of-the-art works rank the results of a keyword query, using traditional IR techniques and exploiting the link structure of the data graph. Then we discuss about related work on the performance of link-based algorithms.

Traditional IR ranking. As we discussed in previous chapters, all major database



(a) DBLPreal

(b) COMSOC k

Figure VI.18: Varying # of keywords

vendors offer tools [2, 3, 1] for keyword search in single attributes of the database. That is, they assign a score to an attribute value according to its relevance to the keyword query. The score is calculated using well known ranking functions from the IR community [47], although their precise formula is not disclosed. Recent works [9, 32, 33, 6] on keyword search on databases, where the result is a tree of objects, either use similar IR techniques [9], or use the simpler boolean semantics [32, 33, 6], where the score of an attribute is 1 (0) if it contains (does not contain) the keywords.

The first shortcoming of these semantics is that they miss objects that are very related to the keywords, although they do not contain them. The second shortcoming is that the traditional IR semantics are unable to meaningfully sort the resulting objects according to their relevance to the keywords. For example, for the query ”XML”, the paper [25] on Quality of Service that uses an XML-based language, would be ranked as high as a classic book on XML [5]. Again, the relevance information is hidden in the link structure of the data graph.

Link-based semantics. The notion of importance has been defined in the context of the Web using PageRank [11], where a global score is assigned to each Web page as we explain in Section VI.A. However, directly applying the PageRank approach in our problem is not suitable as we explain above. HITS [38] employs

mutually dependant computation of two values for each web page: hub value and authority. In contrast to PageRank, it is able to find relevant pages that do not contain the keyword, if they are directly pointed by pages that do. However, HITS does not consider domain-specific link semantics and does not make use of schema information.

Richardson et al. [45] propose an improvement to PageRank, where the random surfer takes into account the relevance of each page to the query when navigating from one page to the other. However, they require that every result contains the keyword, and ignore the case of multiple keywords. Haveliwala [29] proposes a topic-sensitive PageRank, where the topic-specific PageRanks for each page are precomputed and the PageRank value of the most relevant topic is used for each query. Both works apply to the Web and do not address the unique characteristics of structured databases, as we discuss above. Furthermore, they offer no adjusting parameters to calibrate the system according to the specifics of an application.

Recently, the idea of PageRank has been applied to structured databases [26, 34]. XRANK [26] proposes a way to rank XML elements using the link structure of the database. Furthermore, they introduce a notion similar to our ObjectRank transfer edge bounds, to distinguish between containment and IDREF edges. Huang et al. [34] propose a way to rank the tuples of a relational database using PageRank, where connections are determined dynamically by the query workload and not statically by the schema. However, none of these works exploits the link structure to provide keyword-specific ranking. Furthermore, they ignore the schema semantics when computing the scores.

Performance. A set of works [28, 14, 36, 37] have tackled the problem of improving the performance of the original PageRank algorithm. [28, 14] present algorithms to improve the calculation of a global PageRank. Jeh and Widom [36] present a method to efficiently calculate the PageRank values for multiple base sets, by precomputing a set of *partial vectors* which are used in runtime to calculate the

PageRanks. The key idea is to precompute in a compact way the PageRank values for a set of hub pages (we omit the details for brevity reasons), through which most of the random walks pass. Then using these hub PageRanks, calculate in runtime the PageRanks for any base set consisting of nodes in the hub set. However, our problem is simpler, since the number of keywords is much smaller than the number of users in a personalization system. Also, in our case it is not possible to define a set of hub nodes, since any node of the database can be part of a base set.

Chapter VII

Conclusions and Future Work

VII.A Conclusions

In this text we tackled the problem of keyword search in structured and semistructured databases. In particular, given a data graph and a schema graph, we look for subtrees of the data graph that are relevant to the keywords and rank these result-trees according to their relevance. Many works [9, 22, 6, 26] have proposed ways to rank the result-trees using various factors. We presented a framework that captures all previous approaches, which ranks the result-trees according to three factors: (i) the IR scores of the attribute values of the result-trees, (ii) the structure of the result-trees, and (iii) the authority flow between the result-trees and the keywords through the data graph (inspired by PageRank).

We showed that there is an interplay between the last two factors, which were considered unrelated in previous work [29, 45, 22, 9, 6]. In particular, we explain (Section II.D.1) that the number of result trees connecting two nodes is an approximation of the authority flow between them.

Using a combination of the three factors we can capture the semantics of any reasonable ranking criterion. In Section II.C, we present the three most common criteria which are: the global importance of the node of the result-trees, the relevance of the nodes to the keywords, and association between the nodes of

the result-trees. Furthermore, we show how these factors can be combined using a combining function in meaningful ways that allow efficient execution methods.

On the performance level, we present efficient algorithms to produce all or the top-k results of a keyword query. We study two models: the middleware model where the system lies on top of an already operational database system to provide keyword querying, and the dedicated system where we handle the storage of the data and precompute various data to offer real-time response times. In the middleware model (Chapter III), the only interaction to the database system is the creation of full-text indices on the attributes of interest, and submitting and receiving the results of queries. On the other hand, the dedicated system (Chapter V) decides how to store the data in order to allow fast query response times. Also, the authority flow values (Chapter VI) for each keyword are precomputed and stored in an inverted index. Then, in execution time these values are efficiently combined to produce the final result. The execution techniques are thoroughly experimentally evaluated.

Finally, in Chapter IV we presented a novel technique to present the results to the user. In particular, instead of overwhelming the user with a list of all result-trees, we group them by their structure (i.e., by the candidate network that generated them). Then, the user can navigate into these presentation graphs to discover the desirable result.

VII.B Future Work

The semantics of proximity search in database graphs have always been point of debate. The distance between two nodes in terms of number of edges does not always reflect their semantic distance. Also, it is not clear what the equivalent of a document is in a structured database. I plan to create an adjustable framework to accommodate various schemes for the ranking of the results of a keyword query and the semantic distance between the nodes, and evaluate various

configurations in scientific applications, like Geographic Information Systems (GIS) and Bioinformatics. Also, this text has assumed that the only source of semantics of the database is the schema graph. However, other semantic source, like ontologies, may be available and offer an opportunity to improve the quality of the results.

On the performance level, the execution methods in this text were based on a three stage architecture consisting of an inverted index, a CN generator and an execution module. We plan to investigate alternative execution paradigms, which are not necessarily based on evaluating a set of CNs. In particular, we look into various materialization schemes, including materializing path expressions spanning along one or multiple paths. Also, a tighter integration of the inverted index with the execution engine will offer superior performance. Another performance challenge is to efficiently estimate the expected number of results, which is crucial in selecting the right execution method as we explain in Section III.D, and relax the query in case of empty-answer queries.

Furthermore, we have focused on databases with a well defined schema. We plan to investigate how these ideas apply to databases with partial schema description or no schema.

The authority flow factor is a relatively new concept and hence has introduced multiple challenges, on the performance and the semantics level. How can the database schema accelerate the iterative algorithm? What tradeoffs exist between precomputation and on-the-fly execution? Is there a way to incrementally calculate the ObjectRanks in the presence of updates? On the semantic level, a key issue is to find ways to adapt the ObjectRank method to work for databases where the authority flow is not straightforward (as in bibliographic databases). Also, is there a way to automatically adjust the system parameters according to the users' behavior?

Finally, we wish to create a framework to combine the query capabilities of various modules in order to offer greater querying poer to the user. For example, combining an ObjectRank module with a traditional SQL module can answer

questions of the form: Find all authors from UCSD (SQL part) which are authoritative in databases (ObjectRank part). Such a framework imposes challenges on the syntactic, semantic and performance level. In particular, we have to define a way to syntactically combine various query languages, and rank the results of a composite query using the ranking semantics of the individual modules. Also, we need to efficiently combine the results from the modules to get the composite result.

Bibliography

[1] http://msdn.microsoft.com/library/. 2001.

[2] http://technet.oracle.com/products/text/
content.html. 2001.

[3] http://www.ibm.com/software/data/db2/
extenders/textinformation/index.html. 2001.

[4] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. *Addison Wesley*, 1995.

[5] S. Abiteboul, D. Suciu, and P. Buneman. Data on the Web : From Relations to Semistructured Data and Xml. *Morgan Kaufmann Series in Data Management Systems*, 2000.

[6] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. *ICDE*, 2002.

[7] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-Based Keyword Search in Databases. *VLDB*, 2004.

[8] A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. *UCSD Technical Report*, 2001.

[9] G. Bhalotia, C. Nakhey, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[10] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of ICDE*, 2002.

[11] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW Conference*, 1998.

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

[13] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. *Advances in RealTime Systems, pages 225-248, S.H.Son, Prentice Hall*, 1994.

[14] Y. Chen, Q. Gan, and T. Suel. I/O-efficient techniques for computing PageRank. *CIKM*, 2002.

[15] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. *MIT Press*, 1989.

[16] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. *ACM SIGMOD*, 1999.

[17] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *ACM PODS*, 2001.

[18] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, pages 14–23, 1998.

[19] S. J. Finkelstein. Common subexpression analysis in database applications. pages 235–245, 1982.

[20] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[21] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *WWW9 Conference*, 1999.

[22] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. *VLDB*, 1998.

[23] G. H. Golub and C. F. Loan. Matrix Computations. *Johns Hopkins*, 1996.

[24] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. *ICDE*, 1996.

[25] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu. An XML-based Quality of Service Enabling Language for the Web. *Journal of Visual Languages and Computing 13(1): 61-95*, 2002.

[26] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. *ACM SIGMOD*, 2003.

[27] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal 10(4)*, 2001.

[28] T. Haveliwala. Efficient computation of PageRank. *Technical report, Stanford University (http://www.stanford.edu/ taherh/papers/efficient-pr.pdf)*, 1999.

[29] T. Haveliwala. Topic-Sensitive PageRank. *WWW Conference*, 2002.

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

[31] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER:A System for the Efficient Execution of Multi-parametric Ranked Queries. *Proceedings of ACM SIGMOD, Santa Barbara CA*, May 2001.

[32] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. *VLDB*, 2002.

[33] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. *ICDE*, 2003.

[34] A. Huang, Q. Xue, and J. Yang. TupleRank and Implicit Relationship Discovery in Relational Databases. *WAIM*, 2003.

[35] I. Ilyas, W. Aref, and A. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *VLDB*, 2003.

[36] G. Jeh and J. Widom. Scaling Personalized Web Search. *WWW Conference*, 2003.

[37] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Extrapolation Methods for Accelerating PageRank Computations. *WWW Conference*, 2003.

[38] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM 46*, 1999.

[39] J. Kuczynski and H. Wozniakowski. Estimating the Largest Eigenvalue by the Power and Lanczos Algorithms with a Random Start. *SIAM Journal on Matrix Analysis and Applications, v.13 n.4, p.1094-1122, Oct. 1992*.

[40] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with XML and relational. *VLDB*, 2000.

[41] R. Motwani and P. Raghavan. Randomized Algorithms. *Cambridge University Press, United Kingdom*, 1995.

[42] A. Natsev, Y. Chang, J. Smith, C. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. *VLDB*, 2001.

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

[44] J. Plesn'ik. A bound for the Steiner tree problem in graphs. *Math. Slovaca 31*, pages 155–163, 1981.

[45] M. Richardson and P. Domingos. The Intelligent Surfer: Probabilistic Combination of Link and Content Information in PageRank. *Advances in Neural Information Processing Systems 14, MIT Press*, 2002.

[46] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Record*, 29(2):249–260, 2000.

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

[48] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB (Selected Papers)*, pages 47–52, 2001.

[49] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[50] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *VLDB*, 1999.

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

[52] J. A. Storer and T. Szymanski. Data Compression via Textual Substitution. *J.ACM*, 1982.

[53] J. D. Ullman. Principles of Database Systems, 2nd Edition. *Computer Science Press*, 1982.

[54] J. D. Ullman, J. Widom, and H. Garcia-Molina. Database Systems: The Complete Book. *Prentice Hall*, 2001.