Querying Web Metadata: Native Score Management and Text Support in Databases

GÜLTEKIN ÖZSOYOĞLU
Case Western Reserve University
ISMAIL SENGÖR ALTINGÖVDE
Bilkent University
ABDULLAH AL-HAMDANI
Case Western Reserve University
SELMA AYŞE ÖZEL and ÖZGÜR ULUSOY
Bilkent University
and
ZEHRA MERAL ÖZSOYOĞLU
Case Western Reserve University

In this article, we discuss the issues involved in adding a native score management system to object-relational databases, to be used in querying Web metadata (that describes the semantic content of Web resources). The Web metadata model is based on topics (representing entities), relationships among topics (called *metalinks*), and importance scores (sideway values) of topics and metalinks. We extend database relations with scoring functions and importance scores. We add to SQL score-management clauses with well-defined semantics, and propose the sideway-value algebra (SVA), to evaluate the extended SQL queries. SQL extensions and the SVA algebra are illustrated through two Web resources, namely, the DBLP Bibliography and the SIGMOD Anthology.

SQL extensions include clauses for propagating input tuple importance scores to output tuples during query processing, clauses that specify query stopping conditions, threshold predicates (a type of approximate similarity predicates for text comparisons), and user-defined-function-based predicates. The propagated importance scores are then used to rank and return a small number

A preliminary version of this article was published in *Proceedings of the VLDB 2002 Conference*.
This work was supported in part by a joint grant from the National Science Foundation (grant INT-9912229) of the U.S. and TUBITAK (grant 100U024) of Turkey, and the National Science Foundation grants ITR-0312200 and DBI-0218061.
Authors' addresses: G. Özsoyoğlu, A. Al-Hamdani, and Z. M. Özsoyoğlu EECS Department, Case Western Reserve University, Cleveland, OH 44106; email: {tekin,abd,ozsoy}eecs.cwru.edu; I. S. Altingövde, and Ö. Ulusoy, Computer Engineering Department, Bilkent University, Ankara 06800, Turkey; email: {ismail,oulusoy}@cs.bilkent.edu.tr}; S. A. Özel-Özalp (current address): Industrial Engineering Department, Uludag University, Gorukle, Bursa 16059, Turkey; email: ayseozalp@uludag.edu.tr.

of output tuples. The query stopping conditions are propagated to SVA operators during query processing. We show that our SQL extensions are well-defined, meaning that, given a database and a query Q, under any query processing scheme, the output tuples of Q and their importance scores stay the same.

To process the SQL extensions, we discuss two sideway value algebra operators, namely, sideway value algebra join and topic closure, give their implementation algorithms, and report their experimental evaluations.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing; relational databases*; H.2.3 [**Database Management**]: Languages—*Query languages*

General Terms: Algorithms, Languages, Experimentation, Design

Additional Key Words and Phrases: Score management for Web applications

1. INTRODUCTION

This article proposes SQL and database query engine extensions that add a "score management functionality" to DBMSs, where the "scores" of existing database objects are employed to generate scores for query output objects, and to rank them. Score management appears frequently in Web applications. We illustrate with an example.

Example 1.1. Assume that a researcher wants to locate the top-10 most important papers listed at the DBLP Bibliography [Ley] and ACM SIGMOD Anthology sites that are prerequisite papers to understanding the paper "Data Models in Database Management" by E. F. Codd [1980]. At present, this task is performed manually by retrieving the papers cited by Codd's paper iteratively, attaching importance scores to them, and eliminating those that are not in the top-10 prerequisites to understanding the Codd paper, clearly, a time-inefficient process.

Consider a metadata model for DBLP and Anthology sites where "research paper," "Data Models in Database Management," and "E. F. Codd" are topics with importance scores, *Prerequisites* is a relationship among topics (called *associations* in the topic map standard [Biezunski et al. 1999], and here referred to as *topic metalinks*) with importance scores; and for each topic, there are links to Web documents containing "occurrences" of that topic, called *topic sources.* Then, the user can formulate and evaluate the above-specified query using the metadata data model.

In this article, we assume that (i) entities (topics) and relationships (metalinks) (in an object-relational database) have importance scores, and (ii) queries request objects with top-*k* or above-a-given-threshold importance scores. We propose handling query-based score manipulations *natively* within the database query engine, and discuss, for the target area of Web resource querying, a generic (importance) score management component for DBMSs as far as SQL and query processing are concerned.

Score functions appear in the literature in the forms of "scores," "preference values," or "probabilistic values"; we generalize these functions and their evaluations as *sideway functions* and *sideway / importance values*, respectively ("sideway" in the sense that these functions and values are generated not necessarily

Table I. Topics, Metalinks, and Sources Relations in the Metadata Database

Tid	TName	TType	TDomain	Imp
T01	Edward F. Codd	Author	Database	0.9
T08	Data models database management	Paper title	Database	0.8

(a) *Topics* relation

Mid	AuthorId	PaperId
M01	T01	T08

(b) *ResearchPaperOf* metalink relation

Tid	URL
T01	http://www.informatik.uni-trier.de/~ley/db/conf/sigmod/Codd80.html

(c) *Sources* relation

by Web content generators, but by a third party—possibly a data extraction tool). The terms *importance score* and *sideway value* are used interchangeably throughout this article.

We present the score management extensions in a Web database context which we think illustrates best the need for such extensions. We choose as the target area Web resource querying, and, thus, queries have the ability to compare text documents/strings. For Web resource modeling, topics and metalinks constitute metadata (i.e., information about Web resources) representing the advice of data creators, whereas topic sources constitute (URLs to) data, for example, HTML, XML, ps, pdf, text documents. Topics, metalinks, and sources [Biezunski et al. 1999] can be maintained and queried from an object-relational database; the purpose of maintaining topics and metalinks in a database is to be able to pose complex queries, and to quickly locate and rank the associated topic sources on the Web resource.

Example 1.2. Consider the Web resources DBLP Bibliography [Ley] and ACM SIGMOD Anthology. Assume that information about papers (e.g., paper titles, index terms, author names, etc.) in these resources are collected as topics, and stored into the Topics relation, as illustrated in Table I(a). As an example, the tuple with topic id T08 is the 1980 paper of E.F. Codd [1980]. And, the importance of the tuple with Tid T01 is 0.9.

We choose the data model of Table I as our running example for its simplicity; in practice, topics relation is likely to form an inheritance hierarchy with separate authors, articles, etc. relations, each with a large number of additional attributes, etc.. In this article, we assume the following *minimal* data model of metadata, represented as relations of the object-relational model:

—*One Topics(Tid, TName,TType, TDomain, Imp) relation* having topic id, topic name, topic type, topic domain, and topic importance attributes (and possibly other attributes as dictated by the application),
—*One Sources(Tid,URL) relation* with key (Tid, URL) (and possibly other attributes as dictated by the application), and
—*One Metalink relation for each relationship type among topics*, with a metalink id attribute Mid and topic id attributes of topics involved in the relationship (as well as other attributes as dictated by the application). Metalinks

may or may not have importance scores. As an example, *ResearchPaperOf* relation of Table I does not have importance scores; however, *RelatedTo Papers* relation (discussed later) does have importance scores.

These minimal requirements are sufficient to illustrate our SQL and query engine extensions.

Data extraction techniques [Grishman 1997; Agichtein et al. 2000; Agichtein and Gravano 2000, 2003; Brin 1998] can be employed to obtain topics and metalinks with importance scores. We have extracted RelatedToPapers and PrerequisitePapers metalinks for the Anthology (about 15,000) papers [Li 2003; Al-Hamdani 2003], and used them in the experiments of this article. (This article does not describe the data extraction process, and assumes that the metadata is extracted from Web resources and maintained in a database.)

Querying Web metadata stored in a database has two requirements. First, the query language should allow approximate text-similarity comparisons as the Web contains text documents. Second, importance scores of the metadata (i.e., input tuples) need to be used to rank query output topics (tuples), and return either the high-ranking topics above a given threshold, or the top-k highest-ranking topics. We refer to the mechanism that propagates the scores of input topics and metalinks to the output topics and metalinks as the *score management mechanism*. Presently, such mechanisms, if any, are built into applications directly, and outside of database query engines, which is wasteful (each application builds its own score management subsystem) and inefficient (due to the loose coupling between the application and the DBMS as far as the score management is concerned). In this article, we discuss the issues involved in adding a native score management system to a database query engine that allows top-k and threshold-based SQL queries with approximate text-similarity predicates. In more detail, the main contributions of this article are, after extending database relations with sideway value functions and importance scores, to (i) add to SQL text-similarity predicates and score-management clauses with well-defined semantics, (ii) propose an algebra to process the extended SQL queries efficiently, (iii) discuss logical query trees and algebraic optimization for such queries, and (iv) present and evaluate the implementation algorithms for the algebra operators. Below we elaborate more on our approach.

Topic names in the metadata database are arbitrary phrases, which implies the need for efficient approximate text processing and comparison techniques to be incorporated into SQL query processing. We introduce one type of *approximate similarity predicates* into SQL, namely, *threshold predicates*. A threshold predicate compares the text similarity of two text values, and returns true when the evaluated text similarity is above a given threshold; otherwise, it returns false. In addition, a threshold predicate returns an approximate similarity score, which, when the predicate is true, is used for modifying the score of the involved tuple. Thus, threshold predicates are integrated with the score management system, and used for importance score propagation and modification during query processing.

For Web (metadata) databases, the database query engine should return ranked answers to users' queries, necessitating SQL extensions that specify the ranking of output tuples (objects). Our approach is to propagate unambiguously input tuple importance scores of base relations to output tuples, and to use the computed output importance scores in ranking the output tuples. The procedure for importance score propagation and modification within a query is to be specified by the user in the SQL query, and employed by the database system for efficient query processing.

Example 1.3 (*Importance Score Modification*). Consider the metadata of Table I, and assume that the user asks for all authors of database articles with names similar to *E. Codd*. And the similarity between *Edward F. Codd* and *E. Codd* is judged to be 0.7. Then the tuple T01 is returned to the user with the *revised* importance score of $0.9 * 0.7 = 0.63$, where 0.9 is the base importance score of the tuple T01.

To return only the "best" answers in a short time, the SQL query output sizes need to be explicitly controlled by users. For this task, we employ the propagated importance scores of input tuples, and provide two approaches:

(a) For the final output size control, users specify a *ranking threshold* k (i.e., output only the top-ranking k (i.e., top-k) tuples [Carey and Kossmann 1997, 1998; Chaudhuri and Gravano 1999; Chang and Hwang 2002]).
(b) For intermediate output size controls during query evaluation, and for final output size controls, users specify a *sideway value threshold* $\mathrm{V_t}$ (i.e., output all the tuples with importance scores above the threshold $\mathrm{V_t}$).

We refer to these two conditions as *query stopping conditions,* which constitute a user-guided and system-enforced use of importance scores.

We also provide users with the power to modify importance scores in application-dependent ways. For this purpose, UDF (user-defined-function) predicates are defined where, if the predicate is satisfied, output of the UDF modifies the importance scores of tuples.

The existence of importance score modifications and query stopping conditions necessitate the design and evaluation of new join and selection algorithms. In this article, we concentrate on the join evaluation algorithms; selection evaluation algorithms are discussed elsewhere [Al-Hamdani and Özsoyoğlu 2003]

Finally, as illustrated in Example 1.1 with the *prerequisite* relationship, a recursive topic closure operator is useful for user queries. Such an operator serves to retrieve topics related to each other via a particular metalink type, or, more generally, via a regular expression of metalink types.

In more detail, the contributions of this article are as follows:

—Extend SQL with score management and text-similarity-based comparison functionality:
  —clauses that specify unambiguously the propagation and modifications of importance scores of input relations to query output relations in automated ways;
  —clauses that specify query stopping conditions;

—threshold predicates (in the where clause)—if the threshold predicate is satisfied, the output of the similarity score used in the predicate modifies the importance scores of output tuples;
—UDF (user-defined-function) predicates (in the where clause)—if the UDF predicate is satisfied, the output of the UDF modifies the importance scores of output tuples.

Note that the only relational algebra operators that manipulate scores are selection, join, and Cartesian product. SQL queries with aggregate functions and the SQL operator *having* are not discussed here, and constitute future work.

—Show that the above-listed SQL extensions are well-defined, in the sense that, given a database D, the output of a query on D stays the same, regardless of the query processing scheme.
—Present the sideway value algebra (SVA) with two new logical operators, namely, SVA join and topic closure, designed to evaluate the extended SQL queries and to support textual approximate similarity comparisons and recursive closure operations.
—Give implementation algorithms for the SVA join and the SVA topic closure operators. In particular, the SVA join employs a nested loops-based evaluation approach where importance scores and textual approximate similarity among tuple components are exploited for early termination. The closure operator adapts a graph traversal algorithm for its evaluation.
—Experimentally evaluate the SVA join and the SVA topic closure algorithms using real data.

In Section 2, we present the basics of the metadata model and Web queries with examples, and define new SQL extensions. Section 3 introduces the SVA operators for selection, join, and topic closure, and presents logical query trees with these operators. In Section 4, we specify the execution semantics of the extended SQL, and prove that the extended SQL queries are well-defined. Section 5 discusses query processing techniques for the SVA join. In Section 6, we present topic closure evaluation algorithms. Sections 7 and 8 report the experimental SVA join and topic closure results. In Section 9, we review the related work in the literature. Section 10 concludes the article. The electronic Appendix A gives the SVA equivalence rules, while electronic appendix B gives proofs of lemmas and theorems both are available online in the ACM Digital Library.

2. EXAMPLE QUERIES AND SQL EXTENSIONS

2.1 Metadata-Based Web Queries

Below we illustrate the need for score management and approximate text-similarity support in databases, with examples from research paper digital libraries (DBLP and ACM SIGMOD Anthology) as Web resources. However, one can easily envision other Web resource metadata for which a database *natively* supporting score management and text-similarity comparisons would be equally useful. Some examples are (a) Web-based news articles of news agencies, (b) Web-based archeological sites, (c) the Library of Congress Web site [Library], (d) disease-specific (e.g., prostate cancer) Web sites, etc. Moreover, native score management and text-similarity comparison support would also be useful in non-Web-based application frameworks: as mentioned in Carey and Kossmann [1997], there exist applications posing queries with similarity-based ranking requirements to underlying multimedia or text databases.

Example 2.1 (*Threshold Predicates*). Find the topic ids, topic names, and URLs of the 20 highest topic-importance-ranked papers having titles (topic names) with similarity above 0.9 to "query processing". Employ a product-based importance propagation function that uses only topic importance values.

select T.Tid, T.Tname, S.URL
from Topics T, Sources S
where T.TType = "paper title" **and** T.TName $\cong_{(\text{threshold}\,0.9)}$ "query processing"
  and T.Tid = S.Tid
propagate importance as product **function of** T
stop after 20 most important

Topics relation has attributes Tid, TName, TType, and Imp; Sources relation has attributes Tid and URL, storing URLs for the sources of each topic in the Topics table. The predicate "T.TName $\cong_{(\text{threshold}\,0.9)}$"query processing"" states that the topic ("paper title") name of T is similar to "query processing" with similarity above 0.9. We assume that the similarity between a "paper title" and the phrase "query processing" is evaluated by information retrieval techniques, for example, by using the vector space model and the TF-IDF weighting scheme [Salton 1989] (explained in Section 5.1) to represent the topic names. The "propagate importance" clause specifies the importance propagation function for output tuples. In this example, the clause states that the importance scores for output tuples are computed from the importance scores of the base relation Topics, using a "product" function revised with similarities.

Assume that there are three papers with titles "query processing: a survey," "query processing in a P2P environment with extraordinary network bandwidths," and "string processing for C++ applications," and with importance scores 0.9, 0.7, and 1, respectively. Also assume that the similarity function returns the results 0.9, 0.2, and 0.1 for these titles. In this case, the first topic will have the highest score ($0.9 * 0.9 = 0.81$). The second and third topics will have the scores 0.14 ($= 0.7 * 0.2$) and 0.1 ($= 1 * 0.1$), respectively.

The importance score (sideway) function of base relations (denoted by f_{in}) has the range [0, 1]. During SVA operations, for a given output tuple, we materialize the importance score function of the SVA operator, that is, keep it as a (new) column while processing queries.

Example 2.2 (*Join with a User-Defined Function*). Find titles of pairs of conference and journal papers such that journal paper is an *extension* of the conference paper. The user-defined function Extension(T1, T2) returns the similarity of the papers' sources, and we assume that T1 is an extension of T2 if they have at least 50% similarity. Employ a product-based importance propagation function and retrieve the top-100 pairs.

select T1.TName, T2.TName
from Topics T1, Topics T2
where T1.TType = "conference paper title" **and** T2.TType = "journal paper title" **and** Extension(T1.Tid, T2.Tid) $\geq^{sv}$ 0.5
propagate importance as product **function of** T1, T2
stop after 100 **most important**

Here, the predicate "Extension(T1.Tid, T2.Tid) $\geq^{sv}$ 0.5" constitutes a user-defined-function (UDF) predicate (distinguished from an ordinary predicate by the superscript sv). We assume that the UDF function Extension(Tid, Tid) is registered to the DBMS beforehand, and its output modifies the importance scores of output tuples by the value v returned by the UDF if v is greater than 0.5. While evaluating this query, the system propagates and/or modifies the importance scores as specified in the importance propagation clause. In particular, importance scores of selected tuples are determined by multiplying them with the score returned by the UDF. The actual implementation method for evaluating the UDF function, that is, computing content similarity, is "expensive" [Chen 2001; Li 2003], that is, it may require (a) access to actual information resources, such as the above query that needs to do so to compare the contents of two papers, or (b) submitting additional queries to the database.

Example 2.3 (*Topic Closure Query*). Given the relation *Request*(PaperId) containing user-selected paper IDs, the user is interested in finding those ACM SIGMOD Anthology papers that are recursively prerequisites of papers in *Request* with importance values above 0.7. For topic closure, we use a shorthand SQL-like syntax:

select T.TName, S.URL
from Request, Topics T, PrerequisitePapers Prereqs, Sources S
where T.Tid **in** *PrerequisitePapers**(Request,T,{Prereqs}) **and** T.Tid = S.Tid
topic closure importance computation as product **function within a path**
and as max **function among multiple paths**
stop with threshold 0.7

PrerequisitePapers is a metalink type representing the prerequisite paper relationship, and PrerequisitePapers is the relation instance that contains *PrerequisitePapers* metalink instances. * is the Kleene's star. We refer to the predicate "T.Tid **in** *PrerequisitePapers**(Request,T,{Prereqs})" as the *topic closure predicate*. Note that a given paper can have multiple (topic) sources on the Web in terms of a pdf file, a postscript file, an HTML document, or an XML document. Finally, another possible query is to request the top-20 highest importance-valued prerequisite papers of *Request*, which is specified by replacing the *stop with threshold* clause with the *stop after* 20 *most important* clause.

For those database relations that have importance scores (not all may have), we have two ways of specifying tuple (topic/metalink) importance scores: (i) base relation tuples have importance scores explicitly specified as a tuple component (all the examples in this article use this approach), (ii) base relation has an importance (sideway value) function attached, which, when evaluated using a given tuple from the relation, the function returns the importance score of the tuple. Regardless, once the query processing starts, all importance score functions are materialized, and each (intermediate or final output) tuple (object) gets a new tuple component containing the tuple's importance score.

2.2 SQL Extensions

2.2.1 *New Predicates.* As observed from examples of Section 2.1, we employ new SQL *where* clause predicates which, in addition to holding truth values as typical predicates, are also used for importance score modification as dictated by the score propagation clauses (e.g., see Examples 2.1 and 2.2). In this work, we define two particular types of such predicates, namely, threshold predicates and UDF predicates.

The *threshold predicate* is illustrated in Example 2.1 by "T.TName $\cong_{(\text{threshold } 0.9)}$ "query processing"," and has the syntax "X $\cong_{(\text{threshold } t)}$Y" where X and Y are either text-valued variables instantiated by tuple component values or text-valued constants, and t is a real number within the range [0, 1]. The threshold predicate with an instantiation x of X and y of Y is *satisfied* (returns True) if the similarity between x and y (i.e., Sim(x, y) where Sim() is a similarity function) is above the threshold t; otherwise it is not satisfied.

Example 2.4. Consider Example 2.1, in which we modified importance scores with a product function. Then, the importance values of the output tuples for the selection operator with the selection formula "T.TName $\cong_{(\text{threshold } 0.9)}$ "query processing"" is computed as f_{in}* Sim(T.TName, "query processing") where f_{in} "query processing" denotes the importance values of input tuples, and Sim() denotes the similarity function.

User-defined-function (UDF) predicates in SQL queries are illustrated in Example 2.2 by "Extension(T1.Tid, T2.Tid) $\geq^{sv}$ 0.5." The syntax is "UDF θ c" where UDF is a user defined function that returns a real value in [0, 1], θ is a comparison operator from the set $\{<^{sv}, >^{sv}, \leq^{sv}, \geq^{sv}, =^{sv}, \neq^{sv}\}$, and c is a real constant in [0, 1]. The superscript symbol sv in the comparison operator states that the UDF value, when the associated UDF predicate is true, modifies the importance score of the output tuple during query processing.

2.2.2 *New Clauses.* We use the following SQL extensions for score management:

(i) The basic importance propagation clause

"***propagate importance as*** ⟨*ImpAgg*⟩ ***function of*** ⟨*argument list*⟩"

specifies the formula for propagating importance scores of query input relations to the output relation (see Example 2.1). *ImpAgg* is an aggregate function type; in this article, we use the aggregate function *product*. As discussed later in Section 4.3.1 (Rule 4), the function *ImpAgg* is a monotonically decreasing aggregate function, that is, with an enlarged input, it returns a value less than or equal to its previous value. Another aggregate function with this property is *min;* on the other hand, the functions *max* and *numeric-average* do not satisfy this property. The *argument list* is a sublist of relations listed in the *from* clause of the SQL query. In Example 2.1, *ImpAgg* function is product.

(ii) For topic closures, the topic closure (importance computation) clause

"**topic closure importance computation as** ⟨*FPath*⟩ **function within a path**
and as ⟨*FPathMerge*⟩ **function among multiple paths**"

specifies how to compute the derived importance scores of topics encountered during topic closures (see Example 2.3), where *FPath* and *FPathMerge* are aggregate functions. In this article, we use *product* as *FPath.* As discussed later in Section 4.2 (Rule 2), *FPath* is a monotonically decreasing aggregate function of its input. The function *FPathMerge*, on the other hand, is an aggregate function that always produces a value upper-bounded by the maximum value in its input (Rule 3). Thus, possible candidates for *FPathMerge* include *product, max, min,* and *numeric-average.*

(iii) The query stopping clause "**stop after** k **most important**" specifies the ranking (top-k) threshold.

(iv) The query stopping clause "**stop with threshold** V_t" specifies the sideway value threshold.

In this article, all four new SQL clauses as defined above are also allowed in nonaggregate nested SQL subqueries, and have execution semantics similar to ordinary nested SQL queries (as discussed in Section 4). In particular, if the nested subquery is not correlated to the outer query block, it is separately evaluated and its output can be viewed as a materialized input relation for the outer query block. If the nested subquery is correlated to the outer block, whenever the other formulas in the outer block are satisfied, the occurrences of the correlated variables in the nested subquery are replaced by the corresponding variable instantiations of the outer block, and the nested subquery is evaluated as a standalone SQL query several times, that is, once for each correlated variable set instantiation. In the uncorrelated case, the output of the (nonaggregate) nested subquery can be viewed as a materialized relation as far as the outer query evaluation is concerned. In the correlated case, while assigning outer block instantiations to nested subquery variables, the importance scores are also passed to the nested subquery for evaluation. In Section 3.4, we provide an example nested query; in Section 4.3.2, we discuss the query execution semantics for nested subqueries with the query stopping clause *stop after k most important.*

3. SVA OPERATORS FOR EVALUATING EXTENDED SQL QUERIES

For the RA (relational-algebra) operators selection and join, there is an SVA counterpart extended with an output sideway value function f_{out} and the *output threshold* β, which is either the integer-valued *ranking threshold* or the real-valued *sideway value threshold* V_t in the range [0, 1]. And we introduce a new SVA operator, SVA topic closure. In this section, we define and illustrate the

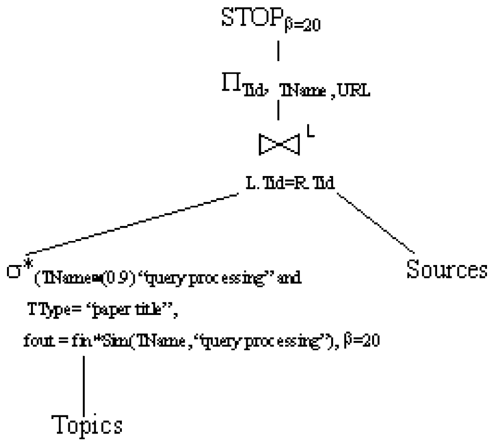


Fig. 1. Logical query tree of Example 2.1.

SVA selection, SVA join, and SVA topic closure operators with example queries and their logical query trees.

In the logical query tree examples discussed next, we use the following notation: operators with superscript * are SVA operators; operators without superscript * are relational algebra (RA) operators; a unary RA operator without * in its superscript carries (if any) into its output tuples the importance scores of its only operand relation; a binary RA operator without a superscript * carries (if any) into its output tuples the importance scores of either its left (hand side) relation or its right (hand side) relation, indicated (if there is a need) by superscript L or R, respectively.

3.1 SVA Selection Operator

In Example 2.1, we gave a query example where topics with names similar to "query processing" over a specified threshold are selected during the query evaluation. The notation $\cong_{(t)}$ in the SVA operator denotes the threshold predicate with the threshold of t.

The logical query tree of Example 2.1 is shown in Figure 1.

Example 3.1. Find the topic IDs of the five highest topic-importance-ranked papers having index terms with similarity to "query processing" above 0.9. Employ min as the importance propagation function that uses all involved importance values.

select distinct Indx.PaperId
from Topics T, IndexedBy Indx
where T.TType = "Index Term" **and** T.TId **in** Indx.TermIdSet **and**
  T.TName $\cong_{(\text{Threshold}\ 0.9)}$ "query processing"
propagate importance as min **function of** T, Indx
stop after 5 **most important**

The logical query tree of Example 3.1 is shown in Figure 2. We assume that *IndexedBy* is a metalink type that specifies the relationship between index

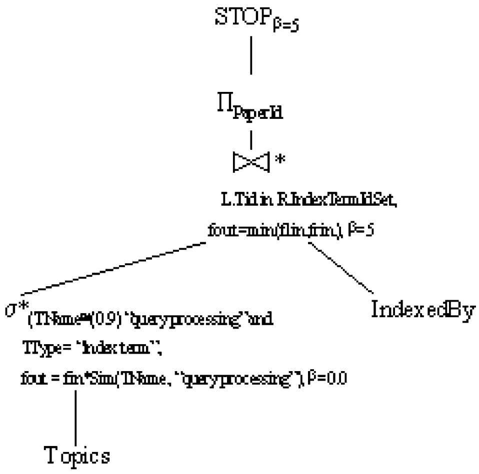


Fig. 2. Logical query tree of Example 3.1.

terms and papers (obtained from keyword/index term list specified in the body of each paper). The signature of the metalink type is *IndexedBy: SetOf Index TermId* $\rightarrow$ *PaperId*. Due to the clause "propagate importance," this query chooses paper ids on the basis of the min of the importance values of index terms (topics) and their *IndexedBy* type metalinks. The function Sim() in Figures 1 and 2 computes the text similarity of two strings, and returns a value in the range [0, 1]. Here, Sim() is used to modify the importance scores of output tuples according to their TName similarity to the string "query processing" (see Table I). The logical query tree shows the SVA selection operator which is denoted as $\sigma^*_{\mathrm{C,\,fout},\beta}(\mathrm{R})$.

Definition (*SVA Selection*). The selection operator $\sigma^*_{\mathrm{C,\,fout},\,\beta}(\mathrm{R})$ takes as input a relation R with a sideway value function $\mathrm{f_{in}}$, a selection condition C, an output sideway value propagation function $\mathrm{f_{out}}$, and the output threshold β where β is either a positive integer k as the ranking threshold, or the real-valued sideway value threshold $\mathrm{V_t}$ in the range [0, 1]. The operator σ^* returns, in decreasing order of output importance scores, either (i) top-k $\mathrm{f_{out}}$-ranking output tuples that satisfy the selection condition C (when β is k), or (ii) all tuples of R with an $\mathrm{f_{out}}$-sideway value greater than $\mathrm{V_t}$ that satisfy the selection condition C (when β is $\mathrm{V_t}$). If the output threshold β is 0.0, it is not applied, that is, the operator is assumed to have no stopping condition and returns all produced tuples.

3.2 SVA Join Operator

Definition (*SVA Join*). The SVA join operator is (L) $\bowtie^*_{\mathrm{A}\theta\mathrm{B,\,fout},\,\beta}$(R) takes as input two relations L and R with sideway value functions fl_{in} and fr_{in}, respectively, a join condition θ on attributes A and B of relations L and R, respectively, a sideway value propagation function $\mathrm{f_{out}}$ for the output tuples, and an output threshold β. The join operator produces joined tuples of L and R with importance scores of output tuples computed as specified by $\mathrm{f_{out}}$ and satisfying the output threshold β.

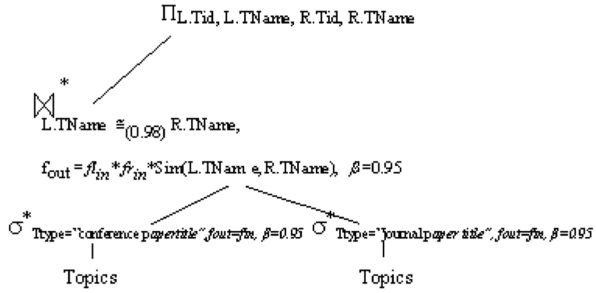


Fig. 3. Logical query tree of Example 3.2.

SVA join in Example 3.1 (Figure 2) is exact, that is, no similarity computations are involved. SVA join in the example below is approximate, with a threshold predicate as a join condition.

Example 3.2 (*Join with a Threshold Predicate*). Assume that topics table allows "journal paper title" and "conference paper title" in topic type field. Find the journal-conference paper pairs with similar titles (i.e., topic name similarity is above 0.98) and return only those pairs that have a derived importance score above 0.95. Employ a product-based importance propagation function that uses all of the involved importance scores.

select T1.Tid, T1.TName, T2.Tid, T2.Tname
from Topics T1, Topics T2
where T1.TType = "journal paper title" **and** T2.TType = "conference paper title" **and** T1.TName $\cong_{(\text{Threshold } 0.98)}$T2.TName
propagate importance as product **function of** T1, T2
stop with threshold 0.95

Note that this query may be posed to see the most important works published both at a conference and in a journal and with highly similar titles.

In Figure 3, the sideway value threshold of 0.95 is propagated to all of the three operators, namely, the two SVA selections and one SVA join. By employing the semantics of propagation to be discussed in Section 4, the similarity score revises the f_{out} value of the joined tuples.

3.3 SVA Topic Closure Operator

Next we define a recursive operator that takes into account the importance scores of its input tuples. Consider the following query and its logical query tree shown in Figure 4.

Example 3.3. Find the topic IDs, titles, and URLs of five highest importance-scored papers such that the selected papers are either (i) papers with titles similar to "Query Evaluation Techniques for Large Databases" with a similarity above 0.85, or (ii) the prerequisites (recursively) of the papers found in (i).

select T2.Tid, T2.TName, S2.URL
from Topics T1, Topics T2, PrerequisitePapers M, Sources S2

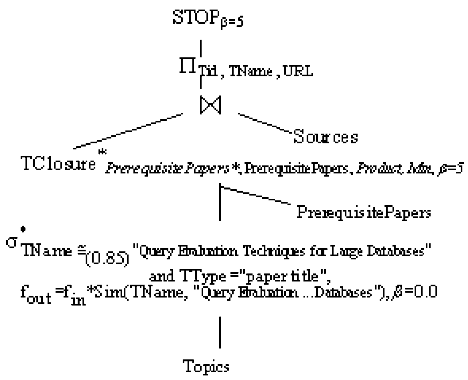


Fig. 4. Logical query tree of Example 3.3.

where T1.TName $\cong_{(\text{Threshold } 0.85)}$ "Query Evaluation Techniques for Large Databases" **and** T1.TType = "PaperTitle" **and**
T2.Tid **in** *PrerequisitePapers**(T1.Tid, T2, {M}) **and** T2.Tid = S2.Tid
propagate importance as product **function of** T1
topic closure importance computation as product **function within a path**
and as min **function among multiple paths**
stop after 5 **most important**

In the above query, prerequisites of the paper "Query Evaluation Techniques for Large Databases" are located recursively by following the metalinks of type *PrerequisitePapers*. For the topic closure predicate evaluation, we introduce the topic closure operator, denoted as $\text{TClosure}^*_{R,\{M\},FPath,FPathMerge,\beta}(\text{X})$, which computes the topic closure X^+ of a set X of topics with respect to a regular expression R of metalink types (and, thus, with respect to the set of axioms characterizing the metalink types in R), a set of metalink relations M, and an output threshold β.

Definition (*Topic Closure*). The operator $\text{TClosure}^*_{R,\{M\},FPath,FPathMerge,\beta}(\text{X})$ takes as input (1) a topic relation, namely, the relation X of topics with a sideway value function f_X, (2) a set of metalink relations M each with a sideway value function f_M, and (3) four parameters: (a) the regular expression R, (b) a path-based "derived" importance score computation function *FPath* that specifies how to compute the derived importance scores of newly reached topics with respect to a single path, (c) the function *FPathMerge* that specifies how to merge the derived importance scores of a given topic obtained through different paths, and (d) the output threshold β. TClosure* computes the closure $\text{X}^+$of X with respect to $\langle \text{R}, \{\text{M}\}, f_X, \{f_M\}, FPath, FPathMerge, \beta\rangle$ where each new topic in the closure is represented as an output tuple, and has a *derived* importance score satisfying the output (ranking or sideway value) threshold β. If the output threshold β is 0.0, it is not applied, that is, the operator is assumed to have no stopping condition and returns all produced tuples.

R is a regular expression of metalink types. For example, the regular expression *PrerequisitePapers*IndexedTerms* finds the index terms in *all* the prerequisite papers (of a given paper topic).

Next we illustrate the notion of paths that satisfy R with an example.

Example 3.4. Let A, B, C, D, and T denote single topics. The metalinks $A \rightarrow^{RelatedTo}$ B, $B \rightarrow^{RelatedTo}$ C, and $C \rightarrow^{RelatedTo}$ T constitute a path P = {A, M_1, B, M_2, C, M_3, T} where all nodes are single topics and all metalinks M_1, M_2, and M_3 have the type *RelatedTo* (i.e., R = *RelatedTo**). As another example, metalinks $AB \rightarrow^{Pre}$ C, $C \rightarrow^{Pre}$ DE, and $DE \rightarrow^{Pre}$ T form a path P = {AB, M_1, C, M_2, DE, M_3, T} that starts with a set of topics AB, followed by a single topic C, then a set of topics DE, and ends with a single topic T. The path P satisfies R = *Prerequisite** since all of its metalinks M_1, M_2, and M_3 are of type *Prerequisite*.

FPath is the derived importance score computation function with respect to a single path. In this article, we use the product function as *FPath*. As an example, assume that the topic t is reached from a topic x in X using a path P = ⟨x m1 a m2 t⟩ where a is a topic with importance score v_a, m1 and m2 are metalinks with importance scores v_{m1} and v_{m2}, respectively, and the metalink types of m1 and m2 satisfy the regular expression R. Assume *FPath* is Product. Then, the derived importance score of t with respect to P, denoted by Imp_d(t, P, R), is computed as the product of importance scores in P that satisfies R, that is, $v_x * v_{m1} * v_a * v_{m2} * v_t$, where v_a and v_t are the importance scores of x and t, respectively. The derived importance score of t, denoted by Imp_d(t, R), is the importance score of t with respect to R and *all* paths leading to t.

The intuition for the semantics of derived topic importance scores is as follows: assume topic t is reached through path P. The derived importance score of t in the closure should be a function of the length and the type of path P, and less than or equal to the importance score of t. As the length of P increases, the derived importance score of t should decrease because t is farther away from (and is *less* related to) the topics in X, the original set of topics listed by the user. Thus, Imp_d(t, P, R) with respect to path P should be a monotonically decreasing function of the length of path P (i.e., *path-monotone*).

FPathMerge is one of Product, NumAve, Min, Max, etc., specifying how to compute the derived importance score Imp_d (t, R) of topic t in X^+ in terms of the Imp_d(t, P, R) scores obtained with respect to each path P.

In Example 3.3, the topic closure importance computation clause specified the use of product function as *FPath,* and min function as *FPathMerge,* as shown in the corresponding query tree.

Finally, we specify the execution semantics of $\text{TClosure}^*_{R,\{M\},FPath,FPathMerge,\beta}(X)$ procedurally as follows:

(a) Locate metalink paths P from a topic in X to a topic t not in X, where P "satisfies" the regular expression R, and compute Imp_d(t, P, R) scores.
(b) Compute the derived importance score of t as sv = Imp_d(t, R), and if sv satisfies the sideway value threshold β then add the new topic t to the closure of X. That is, if β is a positive integer k as the ranking threshold, then sv satisfies β when sv is among the top-*k* output sideway values. If β

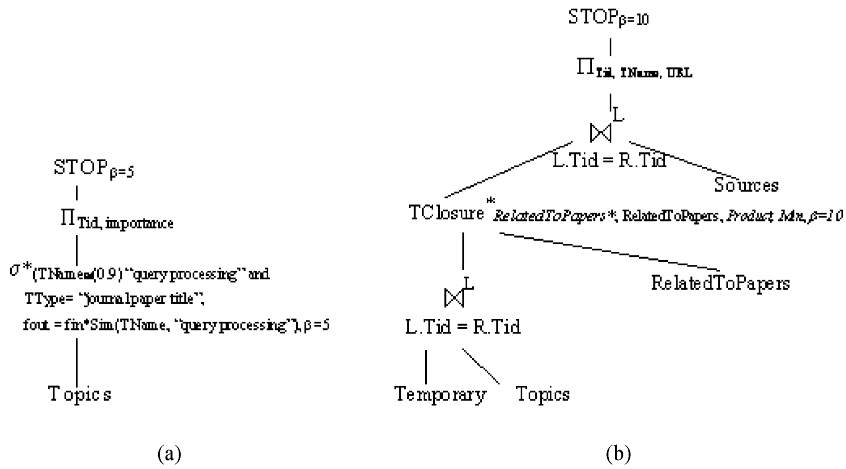


Fig. 5. Logical query tree of Example 3.5: (a) temporary table materialization for inner query, (b) query tree for the outer query.

is the real-valued sideway value threshold V_t in [0, 1], then sv satisfies β when sv $> V_t$.

3.4 SVA Operators in Nested Queries

Consider the nested query example below, and its query tree given in Figure 5.

Example 3.5. Find five highest topic-importance-ranked journal papers having titles similar to "query processing" above 0.9, and then find their 10 most important related papers and the associated URLs. Employ a product-based importance propagation function.

select T2.Tid, T2.Tname, S2.URL
from Topics T1, Topics T2, RelatedToPapers M, Sources S2
where T1.Tid **in** (**select** T.Tid
    from Topics T
    where T.Ttype = "journal paper title" **and**
        T.TName $\cong_{\text{(Threshold 0.9)}}$ "query processing"
    propagate importance as product **function of** T
    stop after 5 **most important**) **and**
  T2.Tid **in** *RelatedToPapers**(T1.Tid, T2,{M}) **and** T2.Tid = S2.Tid
topic closure importance computation as product **function within a path**
    and as min **function among multiple paths**
stop after 10 **most important**

In this example, first the inner query block is evaluated, and an intermediate relation including topic IDs and importance scores (generated automatically) is materialized. Then, this table is used just like base relation with importance scores by the outer query block in a join operation (that implements the set membership), and the final query output is computed. We assume the execution semantics that intermediate relations generated by inner blocks are *implicitly* included in the "propagate importance" clause of outer query, and their scores are propagated. Thus, the importance scores are always propagated from the inner block to the outer block. In the above example, the join semantics enforce that the importance scores of the intermediate relation are propagated, and that T1 and T2 scores are suppressed.

4. EXECUTION SEMANTICS OF THE EXTENDED SQL

Importance score computations (as defined through the SQL extensions of Section 2.2) are functional specifications, superimposed on an SQL query which is logic-based and (mostly) nonprocedural. Therefore, there is a mismatch between functional importance score computations and nonprocedural SQL query specifications. Moreover, importance scores are (a) directly modified by threshold and UDF predicates, and (b) used to choose the final output tuples. Thus, the question arises as to whether the SQL extensions of Section 2.2.2 lead to unambiguous query specifications and unique query outputs.

Definition. An SQL query Q is *well-defined* if, for a given database D, the output of Q is unique.

That is, under any query processing scheme, the output of Q(D) stays the same. In this section, we show that, with the SQL extensions introduced in Section 2.2.2, SQL queries remain well-defined. In other words, input relation importance scores propagate unambiguously and uniquely to intermediate relations and to the final output of the query, which is also unique. This constitutes the specification of query semantics (of the SQL extensions) pertaining to the propagation of importance scores and stopping conditions.

Next, we enumerate the algebra operators used in logical query trees, and discuss which algebra operators modify and propagate importance scores of their operand relations, and how.

(a) *projection, rename, union, set difference, cartesian product, STOP, GROUP-BY operators*: These operators do not have predicates, and, thus, do not modify input tuple scores. However, depending on the needs of the query plan, they may propagate or suppress importance scores of one of their operand relations. Note that *two tuples that are identical in every tuple component but tuple importances are viewed as two distinct tuples*; if they are unioned, both tuples will be in the output. Similarly, projection will materialize importance scores into its output as a column (if the user chooses to retain importance scores in the output of the projection); thus, if two projected tuples are identical in all tuple components except their importance scores, both will be retained in the output of the projection.

(b) *aggregation operators*: When an aggregate function, say, summation on relation R over attribute A (e.g., SUM(R, A)) executes, it aggregates multiple tuples into a single output tuple. Then, the question of how to compute the importance score of the aggregated output tuple from the importance scores of input tuples arises. A simple solution is to attach to each aggregation operator a new "importance score computation function." Such a function would have no constraints, other than the fact that its input is defined in terms of the input tuples of R, and its output needs to be in the range [0, 1]. In this article, we do not deal with aggregate operators.

(c) *join and selection operators*: Through the use of the basic importance propagation clause, and threshold and UDF predicates, these two operators may modify and propagate the importance scores of their operand relations; hence the introduction of the SVA selection and the SVA join operators in Sections 3.1 and 3.2, respectively. In Section 4.1, we define the execution semantics of these two operators, and the conditions under which the query engine decides to generate the appropriate operator (RA or SVA), and then discuss their correctness (i.e., that they are well-defined).

(d) *topic closure operators*: This is a new operator. Through the use of the topic closure importance computation clause and topic closure predicates, this operator also modifies the importance scores of its input tuples; its correctness is discussed in Section 4.2.

The second correctness issue which is orthogonal to the issue of score propagation within a query tree is the propagation of the two query stopping conditions into the SVA operators in the query tree. SVA operators are designed to modify the scores of their input tuples, and the query processing times will be reduced drastically if the query stopping conditions, which are query-wide, can be correctly propagated to SVA operators, and, hence, become "operator-stopping" (i.e., operator-wide) conditions. This is novel since, with the exception of the STOP operator [Carey and Kossmann 1997], none of the algebra operators in the literature contain operator-stopping conditions. In Section 4.3, we study the conditions for propagating the query-wide sideway value threshold V_t and the query-wide ranking threshold (i.e., the top-k condition) into the SVA join, the SVA selection, and the topic closure operators.

4.1 Importance Propagation with Threshold and UDF Predicates

In this section, we assume that SQL queries are extended with threshold predicates, UDF predicates, and the basic importance propagation clause, and discuss the query execution semantics.

Threshold predicates are used by the DBMS as follows. Assume that, during query processing, the threshold predicate P is part of an SVA selection or join operator O, and the evaluation of P for a certain output tuple t of O generates a similarity value v. Then v is used to modify the importance score of t. That is, the similarity values generated by threshold predicates are used in the computation of importance scores for SVA operator output tuples. Consider the *where* clause of an SQL query with threshold predicates. During query processing, those predicates in the *where* clause that compare a single attribute value to a constant, such as the predicate "T.TName$\cong_{(\text{threshold } 0.9)}$ "join algorithms" " will be predicates to an SVA selection operator in the logical query tree, and those predicates that compare two attribute values will be predicates to an SVA join operator in the logical query tree. In both cases, the importance score propagation for the output tuples of the selection or the join operator is extended by the application of a function that involves the value of the similarity function employed in the threshold predicate.

Assume that the SQL query Q uses the basic importance propagation clause (but not the topic closure clause), and has regular, threshold, and UDF predicates (but not topic closure predicates, which are discussed in the next section). Consider

Q: **select**...
from *R, S, T, V*
where...
propagate importance as *product* **function of** R, S

That is, when propagating importance scores of relations R and S for the query at hand, the system will use a product function, and the tuple importance scores of T and V are suppressed, that is, will not be used. We show below that, given an algebra expression E corresponding to query Q on database D, importance scores for the output tuples of E are unambiguously computed and the output of E is unique.

Next we discuss join and selection operators, and the conditions under which the query engine decides to generate an appropriate version (RA or SVA) of the operator. Consider the join operator J in E, with operands E_1 and E_2 that denote either base or intermediate relations, or equivalently the corresponding algebra expressions in E. We evaluate the alternatives:

(i) Neither E_1 nor E_2 is R or S, and neither has at least one of R or S as an argument: in this case, neither of the operands E_1 and E_2 have tuple importance scores (i.e., they are suppressed). Then, the join is an RA join, and the output tuples of the join operator do not have importance scores.

(ii) Only one of E_1 or E_2 is R or S, or has at least one of R or S as an operand, *and* the join condition involves no score-modifying (i.e., threshold or UDF) predicates: let E_1 be the operand involving R or S. Then E_1 has tuple importance scores, and E_2 doesn't. And output tuples of J inherit their importance scores from E_1. In this case, the join operator is an RA join with the provision that it propagates the importance scores of E_1 into the output.

(iii) Only one of E_1 or E_2 is R or S, or has at least one of R, S or both as an operand, *and* the join condition involves either a threshold or UDF predicate, or both: let E_1 be the operand involving R or S (or both). Then E_1 has tuple importance scores, and E_2 doesn't. The output importance scores for the operator J are computed as the product of the tuple importance scores of E_1, similarity values generated by those join predicates that are also threshold predicates (if any), and the values of UDFs for the corresponding UDF predicates (if any). In this case, the join operator is an SVA join.

(iv) E_1 and E_2 are either R and S, respectively, or each has at least one of R or S as an argument: if E_i, $1 \leq i \leq 2$, is R (or S) then the tuple importance scores of E_i are the same as R (or S); otherwise they are computed recursively by considering the operators in E_1 and E_2. The output importance scores for the operator J are computed as the product (i.e., the *ImpAgg* function) of the tuple importance scores of E_1 and E_2, the similarity values generated by those join predicates that are also threshold predicates (if any), and the UDF values of UDF predicates (if any). In this case, the join operator is an SVA join.

Consider the selection operator L in E, with an operand E_1 that denotes either a base or intermediate relation, and a selection condition C applied to E_1. We evaluate the alternatives:

(i) E_1 is either R or S, or has at least one of R or S as an argument, *and* the selection condition C involves either a threshold or UDF predicate, or both: if E_1 is R (or S) then the tuple importance scores of E are the same as R (or S); otherwise they are computed recursively by considering the operators in E_1. The output of the selection operator L contains those tuples that satisfy C. The output tuple importance scores for operator L are computed as the product of the tuple importance scores of E_1, the similarity values of threshold predicates, and the UDF values of UDF predicates. In this case, the selection operator is an SVA selection.

(ii) E_1 is either R or S, or has at least one of R or S as an argument, *and* the selection condition C involves no score-modifying (i.e., threshold or UDF) predicates: if E_1 is R (or S) then the output tuple importance scores of E_1 are the same as R (or S); otherwise they are computed recursively by considering the operators in E_1. The output of the selection operator L contains those tuples that satisfy C. And, output tuples of S inherit their importance scores from E_1. In this case, the selection operator is an RA selection with the provision that it simply propagates the input tuple importance scores into its output tuples.

(iii) E_1 is neither R nor S, and neither has at least one of R or S as an argument: in this case, E_1 has no tuple importance scores (i.e., they are suppressed). Hence, output tuples of the selection operator L do not have importance scores. In this case, the selection operator is an RA selection.

Finally, during the query plan generation for Q, the initial algebra expression E of Q can be transformed into other equivalent algebra expressions. One can specify a set T of algebraic transformations involving RA and SVA operators, and prove that the output of Q stays the same under T. Thus, we have the following lemma.

LEMMA 1. *Nonaggregate SQL queries extended with the basic importance propagation clause, threshold predicates, and UDF predicates are well-defined.*

Hence, we have presented unambiguously the query execution semantics due to a single basic importance propagation clause, and arbitrarily many threshold and UDF predicates.

4.2 Importance Propagation with Topic Closure Predicates

As illustrated in Example 2.3, the topic closure operator is a recursive operator that employs a regular expression (in Example 2.3, the regular expression is "*PrerequisitePapers**") to locate new topics with desired importance scores.

While different metalink types employ different axioms [Özsoyoğlu et al. 2000, 2004], the topic closure operator translates into a "transitive closure-like" operator that traverses over paths of metalinks, and computes importance scores of the newly reached topics that are reached over one or more paths. To compute unambiguously the propagated importance scores of the newly reached topics, we employ the *topic closure* (*importance computation*) clause (as defined in Section 2.2.2(ii)), which is self-explanatory. To have well-defined queries, we use three rules.

Rule 1. Each topic closure predicate is evaluated by a single SVA topic closure operator.

Rule 1 eliminates the use of multiple SVA operators to evaluate a single topic closure predicate, and avoids the specification of topic closure operator interactions within one SQL query.

Definition (*Monotonically Decreasing Function*). Let f be an aggregate function that takes a set of reals in [0, 1] and returns a real in [0, 1]. Let S be a nonempty set of reals in [0, 1] and v be a real in [0, 1]. f is a *monotonically decreasing function* if $f(S \cup \{v\}) \leq f(S)$.

FPath is a (derived) importance score computation function for a topic t reached via a given path.

Rule 2. The function *FPath* defined in the topic closure clause is a monotonically decreasing function.

Rule 2 guarantees that, during the evaluation of the topic closure operator, the search for topics over a metalink path always comes to an end. That is, a topic obtained over a path that includes topic t (and, thus, is "reached" after t is reached) always has a lower propagated importance value than the propagated importance value of t.

FPathMerge function (one of Product, NumAve, Min, Max, etc.) specifies how to compute the (derived) importance score of topic t with respect to multiple paths leading to t.

Rule 3. Assume that the input of *FPathMerge* is the set $S = \{v_1, \ldots, v_n\}$ where v_i is a real in the range [0, 1], $1 \leq i \leq n$. Then *FpathMerge*(S) $\leq$ Max(S).

Rule 3 guarantees that, during topic closure computations, the search for topics over multiple and possibly merging paths comes to an end.

LEMMA 2. *SQL queries extended with a topic closure importance computation clause and employing Rules* 1–3 *are well-defined.*

4.3 Query Stopping Clauses

In Section 2.2.2, we have defined two SQL *query stopping clauses*, namely, threshold and top-*k* clauses, that specify stopping conditions over the query, whose utility is to significantly lower the query processing times. These stopping conditions are enforced by SVA operators (selection, join, and topic closure) in a query tree via the output threshold β.

Next we discuss how the query stopping conditions (i.e., the sideway value threshold V_t or the top-k condition) are propagated to the SVA operators of the logical query tree (i.e., the query execution semantics of the query stopping clauses). In summary, we show below that (i) for the query threshold stopping clause, all SVA operators in the tree enforce the stopping condition, and (ii) for the top-k query stopping clause, only those SVA operators, for which the "score-conservative top-k propagation policy" holds, enforce the stopping condition.

4.3.1 *Stop-with-Threshold Clause.* The *stop-with-threshold* V_t clause directly propagates to all SVA operators of the query *when* the basic importance propagation clause function is a monotonically decreasing aggregate function.

Rule 4. Basic importance propagation clause function f is a monotonically decreasing function.

This rule guarantees that, after propagating $\beta = V_t$ to SVA operators in the query tree, a tuple in the output of a low-level SVA operator and with a score lower than $\beta = V_t$ can be safely eliminated from the output since, if kept in the output of the SVA operator, its score would not increase, and it would not appear in the final query output. Note that the product function used in Section 2.2.2 satisfies Rule 4.

Clearly, such a propagation drastically reduces the intermediate output sizes and query evaluation time. Please note that, before propagating the threshold V_t, we assume that the *stop-with-threshold* V_t clause is enforced with a single STOP operator at the root of the logical query tree with $\beta = V_t$. After propagating β to all SVA operators in the query tree, the STOP operator becomes redundant, and is removed from the query tree.

LEMMA 3. *Consider an SQL query Q with the stop-with-threshold V_t clause and its query tree with a single STOP operator at the root and having $\beta = V_t$. Then, accompanied with Rule* 4*, the threshold V_t propagates to all the SVA operators in the query, and Q stays well-defined.*

Thus, for an extended SQL query with a *stop-with-threshold* V_t clause, all the SVA operators in the corresponding logical query tree inherit the threshold V_t stopping condition, and the query stays well-defined.

4.3.2 *Stop-After-k-Most-Important Clause.* We first discuss the construction of the initial logical query tree. First, a query tree is constructed with RA and SVA operators in which each SVA operator contains a f_{out} function as discussed in Section 4.1, but with no stopping condition, that is, each output threshold β is set to zero. Second, a STOP operator with the top-k threshold (i.e., the query stopping condition) is added as the root. In this section, our goal is to propagate the top-k condition of the STOP operator to lower-level SVA operators as β values whenever possible.

The *stop-after-k-most-important* clause specifies the size of the *final* query output (i.e., the top-k query), and can not easily propagate to intermediate SVA operators of a logical query tree during query processing. This is because such a propagation can prune away some of the intermediate results too early, which may otherwise be included in the final top-k results [Carey and Kossmann 1997, 1998]. On the other hand, applying the top-k stopping condition only at the uppermost SVA operator would eliminate the opportunity of pruning away intermediate tuples, which can never appear in the final output. Here, we revise the conservative strategy proposed by Carey and Kossmann [1997], and propagate the top-k stopping condition only to those SVA operators that do not overprune the intermediate results.

Definition (*Nonreductive Predicate*) [*Carey and Kossmann* 1997]. Consider a predicate p of form x = y where x is an expression computable from an input relation R, and y is an expression involving one or more new relations to be added into the logical query tree. Predicate p is called a *nonreductive predicate with respect to* R if it can be inferred that x cannot be null and, for each x, there exists at least one y satisfying p.

Intuitively, given a relation R as an input to an operator, a nonreductive predicate with respect to R is a predicate that, when used in the operator, returns all the tuples of R in the output of the operator.

Definition (*Score-Conservative Top-k Propagation Policy*). The top-k condition is propagated to an SVA operator V as a stopping condition only when all operators P that directly or indirectly consume the output $O(V)$ tuples of V (i) have nonreductive predicates with respect to $O(V)$, and (ii) propagate tuple importance scores of $O(V)$, but do not further modify them (i.e., each P is either an RA operator, or an SVA operator with $f_{out} = f_{in}$ where f_{in} denotes the scores of $O(V)$).

Condition (i) guarantees that, once a tuple is included in the output of an SVA operator V, it will not be dropped by any other upper-level operators in the logical query tree. Note that condition (i) alone is not adequate for our query evaluation framework due to the score propagation and modification mechanism: assume that an SVA operator which is an ascendant of V revises its input tuple scores by some function f, and a tuple t is already pruned away by V. In this case, it is still possible that t revised by f could have had a higher revised score than the top-k tuples reported in the output $O(V)$ of V, causing a false-drop of tuple t. Thus, condition (ii) is also needed in our policy.

Example 4.1. In Figure 1, the top-k stopping condition is propagated to the SVA selection operator (as it has the β value of 20), due to the score-conservative top-k propagation policy. We assume that every topic has at least one source, and, thus, the join operator above the selection is nonreductive. Moreover, the join is an RA join, which does not revise the scores of tuples returned by the selection, but only propagates them. On the other hand, in Figure 2, the top-k condition is only propagated to the SVA join operator, but not the SVA selection, which has the β value of 0.0. In this case, propagating top-k to SVA selection violates the score-conservative policy since SVA join is both reductive and score-revising. Finally, note that, in Figure 4, the top-k condition (i.e., $\beta = 5$) is propagated to the topic closure operator, according to the score-conservative top-k propagation policy.

Note that the score-conservative top-k propagation policy does not guarantee the uniqueness of the top-k output, as there may be more than one tuple with the same score that are candidates to occur in the top-k output result. That is, there may be n more tuples in the database having the same score with the kth tuple in the output. In this case, for the sake of providing well-defined query results, we include all of these tuples in the final query output and return $(k + n)$ tuples.

A final subtle issue for propagating top-k stopping condition to SVA operators is the need to reapply the *top-k output threshold* β after an SVA operator V in the query tree: assume that the top-k stopping condition of a query Q is propagated to an SVA-operator V for which the score-conservative top-k propagation policy holds. In this case, the operator V will produce at most k tuples and stop, during the query evaluation. But, although the reduction in the intermediate output cardinality is disallowed by our policy, the increase is left unspecified, that is, we have not yet specified the semantics when these k tuples produced by V, say, are joined with more than one tuple in a join later in the query tree. To handle this case, we assume that a STOP operator [Carey and Kossmann 1997], which first sorts its input (if necessary) and then returns the top-k (or $k + n$ as discussed above) tuples, still remains as the outermost query operator regardless of the top-k propagation to SVA operators [Carey and Kossmann 1997]. This guarantees that only the top-k tuples are retained for the final output, but still allows potential reductions in the intermediate output sizes and query evaluation time.

Example 4.2. In Figure 1, the uppermost RA join operator can increase the number of tuples, if each of the k tuples generated by the SVA selection joins with more than one *Sources* tuples. In this case, the STOP operator at the top of the tree guarantees that only the k (or $k + n$) (and no more) tuples are returned as the query output.

We use the following query execution semantics for an extended SQL query with (i) a *stop-after-k-most-important* clause, and (ii) no nested subqueries having the new SQL clauses. The query processor first creates all possible query trees (through applicable algebraic transformations) in which no SVA operator contains the top-k stopping condition. In each query tree, a STOP operator is placed as the root due to the reasons discussed above. The query processor then propagates the top-k condition to the lowest possible SVA operator(s) that satisfies the score-conservative top-k propagation policy, in each query tree. As a result, in each query tree, only such SVA operators will be aware of the top-k condition as an operator-wide stopping condition. The query processor then chooses the query tree with the lowest cost to construct the query plan to execute.

In the case of SQL queries having nested subqueries with their own *stop-after-k-most-important* clauses, the above construction is revised as follows. Consider each subquery independently and materialize it (for subqueries with correlated variables, instantiate the correlated variables when their instantiations satisfy the outer query block). Thus, each subquery can be considered as an independent query with its own top-k condition propagated down the tree properly. Thus, we have the following:

LEMMA 4. *In any SQL query Q, the clause stop-after-k-most-important accompanied with score-conservative top-k propagation policy propagates to SVA operators of Q during query processing, and Q stays well-defined.*

From Lemmas 1–4, we have the following:

THEOREM 1. *SQL queries as defined in Section* 2.2 *and satisfying Rules* 1–4 *are well-defined.*

5. SVA JOIN EVALUATION ALGORITHMS

5.1 Text Similarity Metrics

For those functions that require the similarity comparison $\cong$, we assume that a vector space based similarity model is employed [Salton 1989]. The vector space model first creates a vocabulary (W) of all words (i.e., terms) included in the document collections, and then represents each document with a vector v of |W| terms. The vector entries are real numbers representing term weights. Let v^t denote the vector v element for term t. We use the weighting scheme TF-IDF, which assigns a zero weight for those terms that do not appear in the document, and computes the weights of the other terms using the formula $v^t = (\log(TF_{v,t}) + 1) * \log(IDF_t)$, where $TF_{v,t}$ (term frequency) is the number of occurrences of term t in the document represented by v, and IDF_t is the inverse document frequency that is defined as the ratio of the number of all documents to the number of documents including t. We focus on attributes with short phrases such as topic names. The TF-IDF values are normalized and the similarity of two documents represented with vectors v and u is the cosine of the angle between them, which is defined as Cosine (u, v) $= \sum_{t \text{ in } W} v^t * u^t$.

We assume that term vectors that correspond to string-based attributes of tuples, as well as the vocabulary, are computed a priori. In this section, we assume that vocabulary is small enough to fit in the main memory, whereas all other input and output relations may be arbitrarily large.

Since pipelining is preferable for threshold-based query processing algorithms [Ramakrishnan and Gehrke 2000], and the nested-loop join algorithm does not disrupt pipelining [Graefe 1993], next we discuss block-nested loops-based SVA join algorithms. Moreover, the nested-loop join is appropriate with arbitrary join conditions. A set of nested-loops-based algorithms for processing joins between textual attributes have also been presented in Meng et al. [1998]. We discuss this in Section 9.

In the algorithms below, we assume input relations are sorted in decreasing order of tuple importance scores, and using a sort-merge algorithm might seem like a more reasonable choice than using a block-nested loops join algorithm. However, note that our SVA join condition does not only involve equality; rather, in addition to *score-revising* threshold and UDF predicates, it also involves the computation of an f_{out} function and an *inequality comparison* with the threshold value V_t. In this case, each tuple from one relation will be compared with several tuples from the other relation, and sort-merge algorithm will almost degenerate to nested loops. That is, it is very unlikely that there will be a single scan in

```
Algorithm NLoop_SVT
Input: Sorted Relations R and S wrpt sideway values; f_out() function;
join condition r.A θ s.B; sideway value threshold V_t
Output: {r.s | r∈R and s∈S and f_out(r, s) ≥ V_t and r.A θ s.B}
{i := 1;
while (f_out (r_i, s_1) ≥ V_t and i ≤ |R|)
{ j := 1;
while (f_out (r_i, s_j) ≥ V_t and j ≤ |S|)
{ if r_i.A θ s_j.B then append r_i.s_j to the output;
j++ }     i++ } }
```

Fig. 6. $\mathrm{NLoop}_{\mathrm{SVT}}$ algorithm.

each relation (unless the threshold value is extremely high or the tables are very small, in which case the choice of the join algorithm becomes immaterial) as it is in general sort-merge cases. Thus, the "merge" pointer in the second relation may need to rewind to earlier tuples—perhaps even requiring older blocks to be reread in some cases—per *tuple* in the first relation. Of course, the simple early-termination heuristics discussed below for the nested loops join are equally applicable to sort-merge; but again, performance will not be drastically different from the nested loops approach.

5.2 Nested-Loops-Based Sideway-Value-Threshold Join Algorithms

We now discuss SVA join algorithms that return joined tuples with derived values above a specified sideway value threshold. We assume that the input relations are sorted in decreasing order of tuple importance scores. We sketch two algorithms for join conditions specifying (i) an arbitrary (user-defined) predicate θ over the join attributes, or (ii) an approximate match in terms of the textual similarity of the join attributes.

Definition (*Monotone f_{out}*). Let $\mathrm{sv_t}$ denote the importance score of tuple t. Given relations R and S with tuples r and s, respectively, let $\mathrm{f_{out}}(\mathrm{r}, \mathrm{s})$ denote the importance score of the joined output tuple r.s. Then, $\forall \mathrm{r}_1, \mathrm{r}_2 \in \mathrm{R}$ and $\forall \mathrm{s}_1, \mathrm{s}_2 \in \mathrm{S}$, if $\mathrm{f_{out}}(\mathrm{r}_1, \mathrm{s}_1) \leq \mathrm{f_{out}}(\mathrm{r}_2, \mathrm{s}_2)$ whenever $\mathrm{sv_{r1}} \leq \mathrm{sv_{r2}}$ and $\mathrm{sv_{s1}} \leq \mathrm{sv_{s2}}$, the function $\mathrm{f_{out}}$ is said to be *monotone* with respect to input importance scores of R and S.

Functions product, numeric average, and geometric average are monotone with respect to their input importance scores.

Given a query involving a join with a monotone $\mathrm{f_{out}}$ function, we improve the nested-loop join algorithm by enforcing new stopping conditions while processing the inner and outer loops, as shown in the $\mathrm{NLoop}_{\mathrm{SVT}}$ algorithm in Figure 6. In the $\mathrm{NLoop}_{\mathrm{SVT}}$ algorithm, the inner loop exits whenever the $\mathrm{f_{out}}()$ value of the output tuple r.s is below the threshold $\mathrm{V_t}$, where r is in R and s is in S. Similarly, the outer loop exits at the ith iteration whenever the $\mathrm{f_{out}}()$ value of the output tuple $\mathrm{r_i}.\mathrm{s}_1$ is below the threshold $\mathrm{V_t}$, where $\mathrm{r_i}$ is in R and s_1 is the first tuple in S.

In an ordinary block-nested loops (BNL) join [Ramakrishnan and Gehrke 2000], assuming that the size of R is M pages with p tuples per page, the size of S is N pages with q tuples per page, and the memory has B + 2 buffer pages, we can read B pages of the outer relation R, and scan the inner relation S by using one of the remaining two buffer pages, leaving the last page to collect the output tuples. In this case, the disk access cost of the BNL algorithm is M + (M*N/B) [Ramakrishnan and Gehrke 2000]. In the worst case, the disk access cost of the $\mathrm{NLoop}_{\mathrm{SVT}}$ algorithm is the same as the disk access cost of the BNL algorithm. However, in the expected case, the disk access cost of the $\mathrm{NLoop}_{\mathrm{SVT}}$ algorithm will be reduced depending on how large V_t is. Assume that we revise the allocation of buffer pages as B/2 pages each to the relations R and S; the importance scores in R and S are uniformly distributed, and $f_{out}()$ is the product function, which is monotone. Thus, the tuples in the first B/2 blocks of R have importance scores in the range of $[(1 - B/(2M)), 1]$. Similarly, the tuples in the first B/2 blocks of S have importance scores in the range of $[(1 - B/(2N)), 1]$. During the first outer loop iteration, the inner loop will terminate in the jth iteration when the lowest expected importance score of a join tuple in the buffer is equal to (or ε less than) the sideway value threshold V_t. That is, $(1 - B/(2M)) * (1 - j * B/(2N)) = V_t$. Rearranging the above equality, we have $j = \frac{N}{B/2} * (1 - V_t) - (\frac{2N-B}{2M})$. Assuming $N \gg B$ and $M \approx N$, the above equality reduces to $j = (N/(B/2)) * (1 - V_t)$. That is, in the expected case, for $V_t = 0.9$, the inner loop terminates with 10% of the disk block accesses from S. Since R importance scores are sorted and decreasing in value, for any outerloop tuple of R, S will always be accessed at most for the first $b_S = (N/(B/2)) * (1 - V_t)$ blocks. And, since the above computations are symmetric for R and S, in the expected case, $\mathrm{NLoops}_{\mathrm{SVT}}$ algorithm will terminate with $b_R = (M/(B/2)) * (1 - V_t)$ disk block accesses from R as well. Thus, the expected number E of disk accesses is $E = (B/2) * b_S + (B/2)(b_S - (B/2)) + (B/2)(b_S - 2(B/2)) + \cdots + (B/2)(b_S - (b_R - 1) * (B/2))$. Assuming $b_S = b_R = b$, we have $E = (B/2) * b^2 - (B/2)^2 * ((b^2 - b)/2)$. This, as shown in the experimental results section, is significantly less than the cost of the BNL algorithm.

When the join condition specifies an approximate matching (based on the similarity of the text-valued join attributes being above a given threshold t_{sim}), we cannot directly make use of the similarity function sim(r, s), as it is not monotone, and thus makes f_{out} nonmonotone. However, we can still use the $\mathrm{NLoops}_{\mathrm{SVT}}$ algorithm of Figure 6 with provisions: (a) the functions $f_{out}(r_i, s_1)$ and $f_{out}(r_i, s_j)$ in the outer and the inner while loop conditions are replaced by $sv_{ri} * sv_{s1}$ and $sv_{ri} * sv_{sj}$, respectively, where sv_{ri}, sv_{s1} and sv_{sj} are the importance scores of tuples r_i, s_1 and s_j; (b) in the inner while loop, we check if $f_{out}(r_i, s_j) = sv_{ri} * sv_{sj} * \mathrm{sim}(r_i.A, s_j.B) \geq V_t$ and $\mathrm{sim}(r_i.A, s_j.B) \geq t_{sim}$ where A in R and B in S are the join attributes. If so, the tuple $r_i.s_j$ is output.

Note that, so far, the join algorithm has not employed the similarity function in improving its running time. We now summarize an algorithm that uses the vector-space model and the similarity function in improving the efficiency of the join algorithm.

LEMMA 5. *Let* $\mathbf{u}_r = \langle \mathbf{u}_1 \mathbf{u}_2 \cdots \mathbf{u}_x \rangle$ *be the term vector corresponding to the join attribute* A *of tuple* r *of* R, *where* u_i *represents the weight of the term* i *in* A. *Assume that the filter vector* $\mathbf{f}_S = \langle w_1 \cdots w_x \rangle$ *is created such that each value* w_i *is the max weight of the corresponding term* i *among all vectors of* S. *Then, if*

Algorithm NLoop$_{\text{Sim-SVT}}$
Input: Relations R and S; text-valued join attributes r.A and s.B; Buffers B_S and B_R; sim function sim()=Cosine(); sim threshold t_{sim}

Output: $\{r.s \mid r \in R$ **and** $s \in S$ **and** $f_{out}(r, s) \geq V_t$ **and** Cosine($\mathbf{u_r}$, $\mathbf{u_S}$) $> t_{sim}\}$
1. Sort R by $sv_r *$ Cosine($\mathbf{u_r}$, $\mathbf{f_S}$); Sort S by sv_s;
2. Read tuples from the top of R into a block B_R where, for each r_i in B_R, $sv_{ri} * sv_{s1} *$ Cosine($\mathbf{u_{ri}}$, $\mathbf{f_S}$) $\geq V_t$;
3. Repetitively, read tuples from the top of S into a block B_S, where, for each s_j in B_S, $sv_{r1} * sv_{sj} *$ Cosine($\mathbf{u_{r1}}$, $\mathbf{f_S}$) $\geq V_t$, and compare and join tuples in B_R and B_S:
   for each $r \in B_R$ **do for each** $s \in B_S$ **do**
      if ($sv_r * sv_s *$ Cosine ($\mathbf{u_r}$, $\mathbf{u_s}$)$\geq V_t$ **and** Cosine ($\mathbf{u_r}$, $\mathbf{u_s}$)$\geq t_{sim}$) **then**
         add r.s into the output;
4. Repeat 2-3 until $sv_{ri} * sv_{s1} *$ Cosine($\mathbf{u_{ri}}$, $\mathbf{f_S}$) $< V_t$

Fig. 7. NLoop$_{\text{Sim-SVT}}$ algorithm.

Cosine ($\mathbf{u}_r$, $\mathbf{f}_S$) $< V_t$, *then r cannot be similar to any tuple* s *in* S *with similarity above* V_t.

In this article, the value Cosine ($\mathbf{u}_r$, $\mathbf{f}_S$) is called as the *maximal similarity* of a record r in R to any other record s in S. The maximum value of a term for a given relation is determined while creating the vectors for the tuples, and the filter vector for each relation may be formed as a one-time cost. In Figure 7, we summarize the NLoop$_{\text{Sim-SVT}}$ algorithm which makes use of the sorted order of relations R and S by sv_{r^*} Cosine ($\mathbf{u}_r$, $\mathbf{f}_S$), and sv_s, respectively (also one-time costs). Note that, with both while loop conditions, *false drops* are possible; that is, a tuple r in R and a tuple s in S may satisfy the while loop conditions, only to be eliminated from the output in the if statement within the inner while loop (the if condition tests the values of the actual $f_{out}()$ and sim() functions). On the other hand, while loop conditions do not allow *false dismissals*; that is, a join tuple that is in the output will be added to the output.

5.3 Nested-Loops-Based Ranking-Threshold (Top-*k*) Join Algorithms

It is easy to give an SVA join algorithm with top-*k* output importance scores. Assume that (i) input relations are sorted with respect to importance scores, and (ii) the $f_{out}()$ function is monotone. The algorithm NLoops$_{\text{Top-k}}$ begins in a nested-loop-like manner, and computes the first *k* (but not top *k* yet) joined output tuples, referred to as the "Top-*k*-Set." And the importance score of the kth joined tuple becomes the lower bound (minSV); that is, no tuple with an importance score below this lower bound can be in the top-*k* output. The algorithm proceeds in a nested-loops manner, and updates the lower bound and the current Top-*k*-Set whenever it computes a join output with a new importance score larger than the minimum importance score of Top-*k*-Set.

Similar to the algorithm NLoop$_{\text{Sim-SVT}}$, the algorithmNLoop$_{\text{Top-k}}$ can be revised for a ranking-threshold algorithm NLoop$_{\text{Sim-Top-k}}$ with approximate matching conditions; to save space, it is not presented here.

6. SVA TOPIC CLOSURE ALGORITHM

For the sake of simplicity in presentation, we now summarize TClosure algorithms that compute the topic closure X^+ for the simpler case where the regular expression R is a single metalink type M (however, experimental evaluations of Section 8 use arbitrary regular expressions). Each metalink V $\rightarrow^{M}$ Tid is represented by a tuple in table M, where V is a set of topic identifiers, Tid is a topic identifier, and M is a metalink type.

Definition (*LHS-Decomposability*). *A* metalink of type M is *left-hand side-decomposable* if the axioms of M allow replacing any metalink instance of type M having multiple topics on its left-hand side (LHS) with multiple metalink instances, each having a single topic on its LHS.

As an example, if LHS-decomposibility holds for metalink type M then a metalink instance A, B $\rightarrow$ C, D of type M can be replaced without loss by A $\rightarrow$ C, D of type M and B $\rightarrow$ C, D of type M. We assume in this section that if a metalink type is LHS-decomposable then each metalink with V in the left-hand side is decomposed into multiple metalinks with a single topic in the left-hand side.

Next, we discuss separately the algorithms for sideway value threshold-based and ranking-based topic closures.

6.1 Sideway-Value-Threshold-Based Topic Closure

We create an index MIndex for all metalink instances of all metalink types; and the TClosure algorithm uses only MIndex to find the closure of a given set of topics. We assume that all metalinks are right-hand side decomposed.

MIndex has five attributes: MType, Tid1, Imp(Tid1), ParentList, and ChildList, where MType specifies a metalink type, Tid1 contains the topic identifier of the topic from which the metalink originates, and Imp(Tid1) is the importance score of the topic Tid1. ParentList is a list of topic identifiers of topics from which emanate metalinks of type MType to the topic Tid1. ChildList is a list of triplets ⟨Tid2, Imp(Tid2), Imp(Mid)⟩ where the triplet ⟨Tid2, Imp(Tid2), Imp(Mid)⟩ represents a metalink that has Mid as its metalink identifier, the topic with Tid1 as its antecedent node, the topic with Tid2 as its consequent node, the type MType as its metalink type, Imp(Tid2) as the importance score of the topic with Tid2, and Imp(Mid) as the importance score of the metalink.

The key for MIndex is the two attributes MType and Tid1. Therefore, the MIndex entries with the key (MType, Tid1) contains all metalinks of type MType that have the topic with Tid1 as its antecedent. The entries of MIndex are sorted by (MType, Tid1) so that the metalinks of the same type are together within the index.

Example 6.1. Figure 8 illustrates graphically a Metalinks relation with *RelatedTo* and *Pre(requisite)* metalink instances between five topics. T1 $\rightarrow^{Pre}$ T2 denotes that (learning) T2 is a prerequisite to (learning) T1 [Özsoyoğlu et al. 2004]. Assume that the only axiom for both *RelatedTo* and *Pre* metalink types is transitivity (thus, none of the metalinks in Figure 8 is redundant). Table II shows the tuples of the index MIndex for the Metalinks relation of Figure 8.

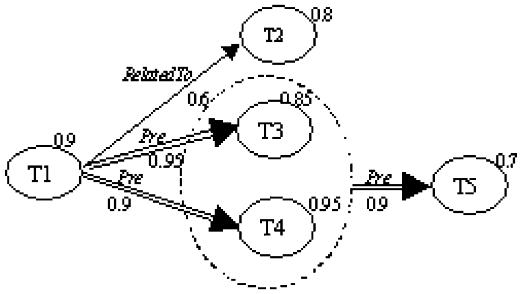


Fig. 8. Graphical representation for non-LHS-decomposable metalink instances in Example 6.1.

Table II. MIndex Table

MType	Tid1	Imp(Tid1)	ParentList	ChildList ⟨Tid2, Imp(Tid_2), Imp(Mid)⟩ triplets
Pre	T1	0.9	{}	⟨T3, 0.85, 0.95⟩, ⟨T4, 0.95, 0.9⟩
Pre	H1	Avg(0.9, 0.95) = 0.925	{T3, T4}	⟨T5, 0.7, 0.9⟩
RelatedTo	T1	0.9	{}	⟨T2, 0.8, 0.6⟩

Table III. HIndex Table for the Non-LHS-Decomposable Metalink in Figure 8

Tid	NodeList ⟨TidList, Hid⟩
T3	⟨{T3, T4}, H1⟩
T4	⟨{T3, T4}, H1⟩

While creating MIndex, for those metalinks that are not LHS-decomposable, we create a second index H(yper)Index to maintain all nodes that are not decomposable; and the topic closure algorithm uses HIndex to compute the closure of a given set of topics. The HIndex table has two attributes Tid and NodeList, where Tid is the topic identifier of a topic t within the nondecomposable node, and NodeList is a list of pairs ⟨TidSet, Hid⟩ where the pair ⟨TidSet, Hid⟩ represents the Tid's of the nondecomposable (hyper) node (which contains Tid), and Hid is a new topic identifier for the node. Table III illustrates HIndex for a nondecomposable node {T3, T4}. We generate a new entry in MIndex for each nondecomposable node with the identifier Hid as its Tid1 value, and with a set of topic IDs that it contains as its "ParentList." For example, in Table II, the entry with the Tid1 value of H1 and the ParentList value of {T3, T4} represents the nondecomposable (hyper) node H1 in the HIndex table.

In this section, to simplify the presentation, we assume that the metalink type M has only the transitivity axiom, and may or may not be LHS-decomposable. And the product function is used to compute *FPath* = $\mathrm{Imp_d}$(t, P, R).

The topic closure of a set X of topics with respect to R as a single metalink type M and a sideway value threshold V_t is computed as follows: for each topic t in the topic closure X^+, we create a triplet of the form ⟨t.Tid, $\mathrm{Imp_d}$(t, R = M), {p | p is a path of type M from a topic or topics in X to t}⟩. We use a set-valued variable DiscoveredTids to contain the topics already in the closure, but not yet checked for paths emanating from them. We construct X^+ by repetitively computing $X^{(0)}$, $X^{(1)}$, ..., $X^{(i)}$ where $1 \leq i$. In the first iteration, for each topic t in X, a triplet $\langle$t.Tid, Imp_d(t, R), {t}$\rangle$ is created in $X^{(1)}$ and the topic identifier Tid of t is added into DiscoveredTids.

In each iteration of the closure algorithm, a topic t1 is removed from DiscoveredTids, and all metalinks that emanate from t1 are visited. A triplet $\langle$t2, Imp_d(t2, R), t2.paths$\rangle$ for the consequent topic t2 of each visited metalink is added into the currently computed topic closure $X^{(i)}$, if the triplet does not exist in $X^{(i)}$. If the triplet exists then new paths are added into t2.paths, and Imp_d(t, R) is recomputed. The topic t2 is then added into DiscoveredTids. If the metalink type M, for which the topic closure is to be computed, is not LHS-decomposable then the algorithm checks if topic t1 is in the LHS of a metalink of type M. The algorithm uses HIndex to find all HIndex entries that contain topic t1 as a member of their LHS set of topics. For each such HIndex entry, if all of its LHS topics are in the currently computed topic closure $X^{(i)}$ then new (hyper)paths are created and new derived importance scores are computed for every metalink that emanates from the HIndex entry. When DiscoveredTids is empty, the algorithm stops, and $X^+ = X^{(i)}$. We refer to this algorithm as the ThresholdTClosure algorithm.

Example 6.2 (*Topic Closure Computation for a LHS-Decomposable Metalink Type*). We use the MIndex instance in Table II. Assume that we want to compute the topic closure for the set X = {T1} with SV threshold $V_t = 0.4$ using the metalink type M = *RelatedTo*. Also, assume that the average function is used for *FPathMerge.* Since X = {T1}, $X^{(1)} = \{\langle T1, 0.9, \{T1\}\rangle\}$ and DiscoveredTids = {T1}. Note that the *RelatedTo* metalink type is LHS-decomposable. In the first iteration, topic T1 is removed from DiscoveredTids. Topic T2 has a path T1.T2, obtained using the metalink $T1(0.9)\rightarrow^{RT(0.6)} T2\,(0.8)$, and its derived importance score is $Imp_d(T2, \mathit{RelatedTo}) = 0.9 * 0.6 * 0.8 = 0.43$. Therefore, the triplet $\langle$T2, 0.43, {T1.T2}$\rangle$ is added into $X^{(1)}$. After the first iteration, $X^{(2)} = \{\langle T1, 0.9, \{T1\}\rangle, \langle T2, 0.43, \{T1.T2\}\rangle\}$ and DiscoveredTids = {T2}. Next, the algorithm terminates since there is no *RelatedTo* metalink emanating from topic T2; therefore, DiscoveredTids becomes empty, and the output of the closure operator is $\{\langle T1, 0.9\rangle, \langle T2, 0.43\rangle\}$. Clearly, if we have more axioms (in addition to transitivity) then the output of the closure will have additional tuples. For example, when the axiom "if $A \rightarrow^{Pre} B$ then $A \rightarrow^{RelatedTo} B$" holds, then all topics will be included into the closure.

Example 6.3 (*Topic Closure Computation for a Non-Left-Hand-Side Decomposable Metalink*). In Figure 8, $\{(T1 \rightarrow^{Pre} T3), (T1 \rightarrow^{Pre} T4)\} \rightarrow^{Pre} T5$ forms a hyperpath of type *Pre* from topic T1 to topic T5. Assume that we want to compute the topic closure for a set of topics X = {T1} with a sideway value threshold $V_t = 0.7$ using the metalink type M = *Pre*, and (a) *FPathMerge* is max, and (b) the geometric average is used to compute the derived importance score of a hypernode. Using the MIndex instance in Table II, we compute X^+ as $\{\langle T1, 0.9\rangle, \langle T3, 0.727\rangle, \langle T4, 0.769\rangle\}$. T_5 is not included into the output because its derived importance score is below the threshold.

During closure computations, a metalink instance (i.e., a tuple in MIndex) can be visited more than once if there are multiple paths to the left-hand-side topic node of the metalink. To avoid visiting the same metalink more than once, we use the parent-child relationship between topics. A topic node with Tid1 is in the parent list of another topic node with Tid2 in the metalink M if there is a metalink Tid1 $\rightarrow^{M}$ Tid2. In the ThresholdTClosure algorithm, we use a set-valued variable PostponedTids to add the restriction that a topic node cannot be "processed" until all nodes in its parent list are processed.

The algorithm ThresholdTClosure needs to maintain *all paths* from the set of input topics X to a given topic instance a in order to compute the derived importance score of a using a generic function. However, some functions, such as max, need to maintain only a single path to compute the derived importance score of a given topic. That is, using the max function, the derived importance score of a topic can be computed by finding the path with the maximum derived importance score. One can give an algorithm ThresholdTClosureMax that does not maintain the path information for any topic, and computes the derived importance score of a topic x by comparing its "current" derived importance score with respect to that of the "currently visited" path P. Clearly, ThresholdTClosureMax is much more efficient than ThresholdTClosure.

6.2 Ranking-Based Topic Closure

We briefly summarize the RankingTClosureMax algorithm that computes the top-k-ranked topic closure using product as *FPath* and max as *FPathMerge.* The algorithm finds the topics with the k highest derived importance scores in the topic closure of a set X of input topics. It first computes the initial candidate top-k-ranked topics from the input topics X. Then, in each iteration i, it extracts the ith top-ranked topic from the current $k - \mathrm{i} + 1$ candidate top-ranked topics and updates the current candidate topics by processing all emanating metalinks from the ith topic. Therefore, the algorithm needs k iterations in order to compute the top-k-ranked topic closure of a set X of input topics.

The RankingTClosureMax algorithm maintains two lists X+ and CandidateTopics of size at most k. The algorithm requires at most $\Omega(k * |\mathrm{X}|)$ time to compute the initial CandidateTopics list, where |X| is the size of the input topic set X. Then, the algorithm iterates k times in order to compute the top-k-ranked topic closure, and, in each iteration, it finds the next top k topics and updates the CandidateTopics list by applying the metalinks that emanate from a given top-k topic.

7. EXPERIMENTAL RESULTS: EVALUATING THE SVA JOIN OPERATOR

To evaluate the four SVA-join algorithms discussed in Sections 5.2 and 5.3, we first extracted all the titles of journal and conference papers from the DBLP [Ley] data set into two different files, R and S—R with more than 91,000 journal paper titles (12 Mbytes), and S with more than 132,000 conference paper titles (18 Mbytes). Next, we eliminated the stopwords (i.e., removed words like *the*, *a*, *of*, etc.) from the text in each title, stemmed them, and created the word list (vocabulary) for the whole collection (including about 43,000 words). The word list was kept in the main memory. Then, we created the vectors for each record of R and S, which were added to paper title records in files R and S.

Topic importance scores for papers were computed based on the rankings of the journals or conferences in which they appeared. For this purpose, we used the ranking list provided at CiteSeer [2003]. We split this list into 10 bins, giving the importance score 1 to those venues ranked in the top 10%, score 0.9 to those ranked in the 11–20% slice, and so on. It turned out that some of the publication venues encountered in the DBLP data set were not found in CiteSeer's list. These were assigned the importance score 0.6, since the average impact estimation score of DBLP venues that appear in CiteSeer list falls into the bin with the score 0.6. Note that we may perhaps have overestimated the importance scores of these venues (and the papers published in them), as these unlisted venues are potentially less-known and less-important ones. As supporting evidence for this claim, we found out that of 210,000 journal/conference papers in our test set, less than 5% were published in those venues. As another observation, a considerable number of the papers listed in DBLP are published in the venues ranked in the top 10% of CiteSeer, resulting in the score attachment of 1. Thus, our importance score assignment was not uniform, but depended on the properties of real data published at the DBLP site. As a final remark, it could be argued that not all papers published at the same venue have the same importance scores; however, our intention in this section is not to develop a method for measuring paper importance, but rather to provide experimental evaluation of a data set that approximately fits to real-life application constraints.

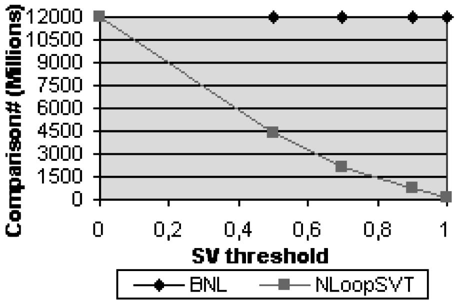
Below, we provide the experimental evaluations of the SVA-join algorithms in terms of the number of comparisons for a given query. The number of comparisons gives an idea about the number of tuples read from each relation. Results involving disk-accesses and execution times were clearly symmetric with the number of comparisons made, and not reported here.

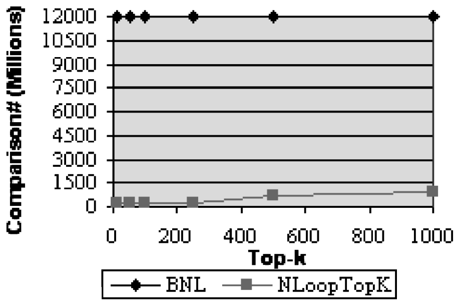
All experiments were performed on a dual-processor Pentium III PC with 1-GB main memory running WindowsNT 4.0. The input and output buffer sizes held 10,000 tuples. The algorithms were implemented in C programming language.

7.1 Evaluating $\mathrm{NLoop}_{\mathrm{SVT}}$ and $\mathrm{NLoop}_{\mathrm{TOP\text{-}K}}$

These algorithms join tuples of R and S on the basis of an arbitrary join condition (predicate) θ, and return the joined tuples that are over a given threshold V_t or ranked in the top-k results. For the following experiments, $f_{out}()$ is specified as the product of the importance scores of joined tuples. We assume that join condition θ is a user-defined function that requires further (and presumably expensive) processing of the tuples, as illustrated in Section 3.2. For instance, such a function may state that a conference paper tuple is to be joined with a journal paper if they have at least one author in common and the conference paper is published at most 2 years before the journal paper. Clearly, this predicate can be specified as a UDF (syntax omitted to save space).

To evaluate a join with an arbitrary condition θ, an ordinary block-nested loops algorithm compares each and every tuple, computes the importance scores for those tuples satisfying the user-defined function, and finally retrieves the ones that are above the specified threshold or in the specified top-k set. On the





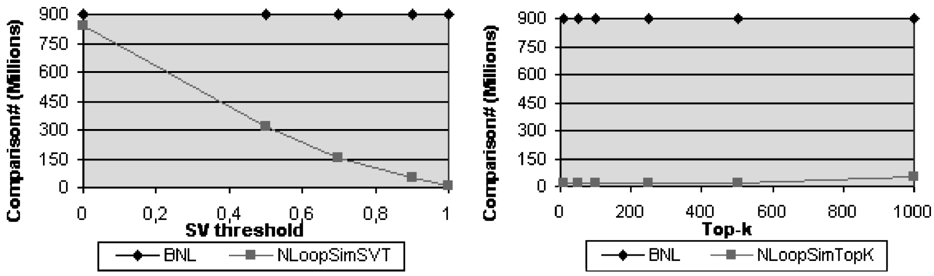
Figs. 9 and 10. Performance values of BNL versus $\text{NLoop}_{\text{SVT}}$ and $\text{NLoop}_{\text{Top-k}}$, respectively.

other hand, $\text{NLoop}_{\text{SVT}}$ and $\text{NLoop}_{\text{Top-k}}$ evaluate the arbitrary predicate only for those tuples with derived importance scores that satisfy the query constraints. In Figures 9 and 10, we demonstrate the performance of these algorithms, compared against the "blind" BNL approach. Note that the savings of the proposed algorithms increase as the SV threshold value increases or, inversely, as the k value decreases. For instance, when the SV threshold value is 0.9, the number of tuple comparisons performed by $\text{NLoop}_{\text{SVT}}$ is approximately 600 million, 1/20 of the BNL approach, which makes 12 billion comparisons. For this case, $\text{NLoop}_{\text{SVT}}$ reads only 27% of R and 60% of S from the disk, whereas BNL reads all tuples of the relations. The saving in terms of execution time also matches well with the 1/20 ratio of tuple comparisons, that is, 3 min versus 1 h. Note that the percentages of tuple readings from each relation show that tuples with high importance scores dominate the DBLP data set, as we have mentioned before, and savings would increase for those cases where only a few tuples can exceed the SV threshold.

7.2 Evaluating $\text{NLoop}_{\text{Sim-SVT}}$ and $\text{NLoop}_{\text{Sim-Top-k}}$

These algorithms perform similarity-based (approximate) joins. In the following experiments, the tuples of R and S are joined if their titles are similar with a similarity value greater than a specified threshold (90%). In this case, $f_{out}()$ is specified as the product of the importance scores of joined tuples and this derived value is further multiplied with the similarity value of tuples, obtained using the cosine similarity measure.

Figures 11 and 12 illustrate the performance superiority of $\text{NLoop}_{\text{Sim-SVT}}$ and $\text{NLoop}_{\text{Sim-Top-k}}$with respect to the BNL. Note that, as discussed before, a blind BNL would compare all pairs, leading to almost 12 billion tuple comparisons. For the special case of similarity-based predicates, we employed an inverted index while computing the similarity of the tuples that were read and buffered (in a similar fashion to the probing phase of hash-join [Ramakrishnan and Gehrke 2000]). More specifically, we created an in-memory inverted index [Salton 1989] for the tuples of outer relation on the fly, and compared tuples of inner relation that only had common words in their titles. Thus, for all of the algorithms, the results reported in the figures indicate the number of accesses to the in-memory inverted index during the comparison, that is, BNL accesses to the index 900 million times, also implying the same number of similarity comparison



Figs. 11 and 12. Performance values of BNL versus $\text{NLoop}_{\text{Sim-SVT}}$ and $\text{NLoop}_{\text{Sim-Top-k}}$ algorithms, respectively.

computations, although it read the blocks of the both relations entirely several times. We observe that SVA algorithms again considerably reduce the cost of the join operation. For instance, to retrieve tuple pairs with titles that are 90% similar and have a derived importance score greater than 0.9, BNL achieves a total of 900 million computations, whereas $\text{NLoop}_{\text{Sim-SVT}}$ makes only 50 million computations. This improvement is due to the fact that similarity-based algorithms are tailored to exploit the vector-space model to its greatest extent.

To summarize, for arbitrary predicates and monotone SV functions, algorithms $\text{NLoop}_{\text{SVT}}$ and $\text{NLoop}_{\text{Top-k}}$improve the performance of BNL considerably. For the special case of text similarity-based joins, the algorithms were further optimized (e.g., by using the maximal similarity filter heuristic), and more gains were obtained.

8. EXPERIMENTAL RESULTS: EVALUATING THE SVA TOPIC CLOSURE OPERATOR

We evaluated the performance of the TClosure algorithms using all the articles in the ACM SIGMOD Anthology between 1969 and 2001. All of the articles, available as pdf files, were parsed, and indices were constructed and used to extract metalinks between papers, such as the *RelatedTo*, *Prerequisite*, and *WrittenBy* metalinks. In Al-Hamdani [2003], we provided a more detailed description of the metadata extraction process from the ACM Anthology.

Using topics and metalinks, disk-based index files were constructed. And, in order to efficiently retrieve tuples from two index files (MIndex and HIndex), a memory-based sparse index table was employed. In implementations of topic closure algorithms, we used *max* as the *FPathMerge* function. We evaluated the performances of the Threshold-based and Top-k-based TClosure algorithms in terms of the number of disk accesses and the size of the output result X^+.

We employed a finite state automaton (FSA) that corresponded to a given regular expression R. As an example, the FSA in Figure 13 corresponds to the regular expression $R = PRE^*.RT.RT^*$ (where *PRE* and *RT* are *Prerequisite* and *RelatedTo* metalinks, respectively).

8.1 Data Generation

All ACM SIGMOD Anthology articles (14,891 papers) were converted from pdf files into plain text files. Then, DBLP bibliography information

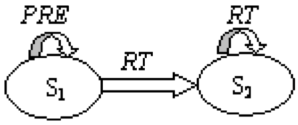


Fig. 13. FSA for regular expression R = $PRE^*.RT.RT^*$.

[Ley] was used to extracted the titles, authors, publication venue (conference or journal), and the publication year for each paper in the ACM Anthology. We also extracted the abstract, index terms, body, and references for each paper using its text file. The TF-IDF vectors were used to represent each component of each paper (i.e., the title, abstract, index terms, and body) and to created the corresponding index files. We also created index files for authors, references, and the publication venue of the papers.

8.1.1 *Topic Extraction.* We extracted two types of topics, papers and authors, and computed their importance scores.

(1) *Paper importances.* The importance score of a given paper can be computed in multiple ways, such as the following:
   (a) Publications that get referenced by highly "important" papers are more important (residual effect). *PageRank* [Brin and Page 1998] algorithm can be used to recursively compute the importance scores of papers using the importance scores of papers that cite them.
   (b) The notion of hubs and authorities (i.e., the HITS algorithm of Kleinberg [1998, 1999]) among papers can be used to compute importances of papers.
   (c) Citation count: how many times a paper is cited by other important papers.
   (d) Publication venue: for example, SIGMOD versus CIKM. The importance score of a conference or journal influences the importance of a paper.
   (e) Temporal distributions of citations with respect to duration.
   (f) Citation venue: for example, survey journal versus research paper.
   (g) Citations by "important" authors' work are more significant.
   (h) Importance of an author influences importances of his/her papers.

   In this article, we compute importance scores of papers using (i) citation counts, (ii) publication venue, and (iii) importance scores of the most important papers that refer to the given paper.

   (i) *Citation count.* For a given paper P, let *CitationCount*(P) be the number of times paper P is cited by other papers. Using the number of citations, paper P is as important as those papers that have the same number of citations and more important than the papers that have fewer citations. Now, let *PapersWithCitations*(i) be the number of papers that were cited i times. We computed the importance of a given paper P with respect to its citation count as follows:

$$ImpPaper_{CitationCount}(P) = \sqrt{\frac{\sum_{i=0}^{CitationCount(P)} \text{PapersWithCitations(i)}}{\text{No. of papers}}}$$

(ii) *Publication venue.* The importance score of a given conference or journal was computed using the total number of papers it had and the total number of citations to its papers. We computed the unnormalized conference importance scores using the following formula:

$$ImpConf_U(\mathrm{V}) = (\#\text{citations of } Conference\ V)/\sqrt{\#\text{papers in } Conference V}.$$

Let $ImpConf_{Max}$ be the unnormalized importance score of a conference with the highest unnormalized importance score. By applying the *ConfMinImp factor*, where $0 \leq ConfMinImp \leq$, we found the importance scores for a given conference or a journal as follows:

$$\begin{aligned} ImpConf(\mathrm{V}) &= ConfMinImp + (1.0 - ConfMinImp) \\ &\quad * \sqrt{ImpConf_{\mathrm{U}}(\mathrm{V})/ConfImp_{\mathrm{Max}}} \quad . \end{aligned}$$

We used $ConfMinImp = 0.4$ in the experiments.

(iii) *Adding the citation effect of the most important citation*. For a given paper P in conference V, let P_{maxcit} be any paper that cites paper P with the highest importance score. We computed the importance score of a paper P using

$$\begin{aligned} ImpPaper(P) &= (1 - MaxCitFactor) * [(ConfFactor * ImpConf(\mathrm{V})) \\ &\quad + (1.0 - ConfFactor) * ImpPaper_{CitationCount}(\mathrm{P})] \\ &\quad + MaxCitFactor * Imp(P_{maxCit}), \end{aligned}$$

where $0 \leq MaxCitFactor \leq 1.0$ and $0 \leq ConfFactor \leq 1.0$. In the experiments, we used $MaxCitFactor = 0.2$ and $ConfFactor = 0.7$.

(2) *Author importances.* The importance score for an author can be computed in multiple ways:
(a) the most important paper of the author,
(b) the weighted average of the most important k papers of the author;
(c) the weighted average of the most important m% papers of the author;
(d) the weighted average of the most important papers of the author in every y years.

We computed the importance score of an author using 20% of his/her most important papers. For the ACM Anthology, the importance scores of (a) 106 conferences, journals, and books, (b) 14,891 papers, and (c) 13,208 authors, were computed and stored in files. The papers were stored in a file of 222-kB size, and the authors were stored in a file of 198-kB size.

8.1.2 *Metalink Extraction.* Three types of metalinks and their importance scores were extracted, namely, *RelatedTo*, *Prerequisite*, and *WrittenBy*.

(1) *RelatedTo metalink instance extraction.* A paper $\mathrm{P_i}$ is related to a paper $\mathrm{P_j}$ if the similarity $Sim(\mathrm{P_i}, \mathrm{P_j})$ is above a given threshold value V_t (in the experiment, we used V_t a value of 0.4). We computed the similarity between two papers using a weighted function of their title similarity Sim_{Title}, index terms similarity $Sim_{IndexTerms}$, abstract similarity $Sim_{Abstract}$, body similarity Sim_{Body}, author similarity Sim_{Author}, and references similarity $Sim_{References}$.

We used the TF-IDF vectors with the *cosine* similarity measure [Salton 1989] to compute the similarities between two paper's titles, abstracts, index terms, and bodies. Each of these similarities was referred to as a *similarity factor*. We first removed the stopping words from the terms of a similarity factor, and then used Porter's [1980] algorithm to stem the terms.

We computed the author similarity between two papers using the "*Level-0-author-overlap*" relationship (i.e., common authors between two papers) and the "*Level-1-author-overlap*" relationship (i.e., two different authors, each of different papers $\mathrm{P_i}$ and $\mathrm{P_j}$, are coauthors in a third paper $\mathrm{P_k}$). We used the following formula to compute the author similarity between two papers:

$$\begin{aligned} Sim_{Author}(\mathrm{P_i}, \mathrm{P_j}) = {} & L0Weight * Sim_{Level\text{-}0\text{-}Author}(\mathrm{P_i}, \mathrm{P_j}) \\ & + (1 - L0Weight) Sim_{Level\text{-}1\text{-}Author}(\mathrm{P_i}, \mathrm{P_j}), \end{aligned}$$

where $0 \leq L0Weight \leq 1$.

The reference similarity between two papers $\mathrm{P_i}$ and $\mathrm{P_j}$ was computed using the *bibliographic coupling* (the number of common citations between the two papers [Kessler 1963]) and *cocitation* (cocitation frequency with which two papers appear as citations in the same document [Small 1973]) between the two papers. We computed the reference similarity as follows:

$$\begin{aligned} Sim_{References}(\mathrm{P_i}, \mathrm{P_j}) = {} & BibWeight * Sim_{bib}(\mathrm{P_i}, \mathrm{P_j}) \\ & + (1 - BibWeight)\, Sim_{coc}(\mathrm{P_i}, \mathrm{P_j}), \end{aligned}$$

where $0 \leq BibWeight \leq 1$. In the experiment, we used *L0Weight* and *BibWeight* values of 0.7 *and* 0.6, respectively.

Finally, we used the following formula to compute the importance score of the *RelatedTo* metalink instance between two papers $\mathrm{P_i}$ and $\mathrm{P_j}$:

$$\begin{aligned} & Imp\ (RelatedTo(\mathrm{P_i}, \mathrm{P_j})) = Sim(\mathrm{P_i}, \mathrm{P_j}) \\ & = W_{Title} * Sim_{Title}(\mathrm{P_i}, \mathrm{P_j}) + W_{IndexTerms} * Sim_{IndexTerms}(\mathrm{P_i}, \mathrm{P_j}) \\ & \quad + W_{Abstract} * Sim_{Abstract}(\mathrm{P_i}, \mathrm{P_j}) + W_{Body} * Sim_{Body}(\mathrm{P_i}, \mathrm{P_j}) \\ & \quad + W_{Author} * Sim_{Author}(\mathrm{P_i}, \mathrm{P_j}) \\ & \quad + W_{References} * Sim_{References}(\mathrm{P_i}, \mathrm{P_j}), \end{aligned}$$

where $W_{Title} + W_{IndexTerms} + W_{Abstract} + W_{Body} + W_{Author} + W_{References} = 1.0$.

There is also the issue of choosing the right values for weights $W_{IndexTerms}$, $W_{Abstract}$, W_{Body}, W_{Author}, and $W_{References}$. In Li [2003], an experiment was performed to locate the similarity weights that produce the highest precision queries using 1000 papers. The experiment showed that the similarity factor weights with the highest precision were $W_{Title} = 0.143225$,

$W_{IndexTerms} = 0.0607289$, $W_{Abstract} = 0.183921$, $W_{Body} = 0.151375$, $W_{Author} = 0.202429$, and $W_{References} = 0.2583211$. Therefore, we used these weights in computing the importance scores for *RelatedTo* metalinks.

We normalized the similarity values for each similarity factor, say F (e.g., F = title), using the maximum similarity $Sim_{Fmax}(\mathrm{P_i})$ between a paper $\mathrm{P_i}$ and all other papers. *RelatedTo* metalink is reflexive; therefore, for any two papers $\mathrm{P_i}$ and $\mathrm{P_j}$, *Imp*(*RelatedTo*($\mathrm{P_i}$, $\mathrm{P_j}$)) = *Imp*(*RelatedTo*($\mathrm{P_j}$, $\mathrm{P_i}$)). To maintain the reflexivity property, we normalized the similarity values for a given similarity factor Sim_F between papers $\mathrm{P_i}$ and $\mathrm{P_j}$ using the minimum of $Sim_{Fmax}(\mathrm{P_i})$ and $Sim_{Fmax}(\mathrm{P_j})$. Thus,

$$Sim_{F_{Normalized}}(\mathrm{P_i}, \mathrm{P_j}) = Sim_F(\mathrm{P_i}, \mathrm{P_j})/min(Sim_{Fmax}(\mathrm{P_i}), Sim_{Fmax}(\mathrm{P_j})).$$

(2) *Prerequisite metalink instance extraction.* We used the citation information to extract *Prerequisite* metalinks. A paper $\mathrm{P_i}$ is a prerequisite to a paper $\mathrm{P_j}$, written as *Pre*($\mathrm{P_i}$, $\mathrm{P_j}$), if paper $\mathrm{P_i}$ appears in the references of paper $\mathrm{P_j}$. We used the occurrences of the cited papers to compute the importance scores for their prerequisite metalinks. Let $\mathrm{O_{max}}(\mathrm{P_j})$ be the number of occurrences of the most cited reference in the body of a given paper $\mathrm{P_j}$, and $\mathrm{O}(\mathrm{P_i}, \mathrm{P_j})$ be the number of occurrences of a reference $\mathrm{P_i}$ in the body of paper $\mathrm{P_j}$. Then, the importance score for the prerequisite metalink instance *Pre*($\mathrm{P_i}$, $\mathrm{P_j}$) was computed using the formula

$$Imp(Pre(\mathrm{P_i}, \mathrm{P_j})) = (O(\mathrm{P_i}, \mathrm{P_j}) + 1)/(O_{\max}(\mathrm{P_j}) + 1). \tag{1}$$

We added 1 to the number of occurrences and to the maximum occurrences so that all the importance scores were greater than zero. Another alternative was to compute the importance scores using the following formula:

$$Imp(Pre(\mathrm{P_i}, \mathrm{P_j})) = MinPreFactor + (1 - MinPreFactor) * O(\mathrm{P_i}, \mathrm{P_j})/O_{\max}(\mathrm{P_j}), \tag{2}$$

where $0 \leq MinPreFactor \leq 1$.

In our implementation, we evaluated both Formulas (1) and (2).

(3) *WrittenBy metalink instance extraction.* One can construct *WrittenBy* metalink importance scores using the importance scores of the authors of papers. However, in the experiments, we assumed that the *WrittenBy* metalink type did not have importance scores (or, more correctly, for each paper $\mathrm{P_i}$ and author $\mathrm{A_j}$, *Imp*(*WrittenBy*, $\mathrm{P_i}$, $\mathrm{A_j}$) was assumed to be 1.0).

Using the papers in the ACM Anthology [ACM SIGMOD Anthology], we extracted 40,486 *RelatedTo* metalinks, 30,772 *Prerequisite* metalinks, and 34,244 *WrittenBy* metalinks. The total size of the metalink file was 1.8 MB.

8.2 Metalink Index Generation

We created the index file *MIndex* with the key ⟨Tid, MType⟩ for all metalink types, stored as a paged file on secondary storage. Each *MIndex* page contained data about metalinks of the same type MType (*MIndex* is ordered by topics identifiers Tids), and was of size at most *PageSize* (we used *PageSize* of 1 KB). Index entries for the metalinks for a given key ⟨Tid, MType⟩ were maintained in

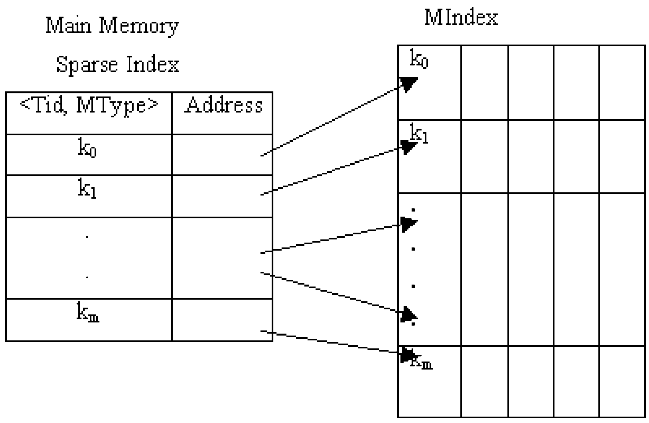


Fig. 14. Disk-based index table.

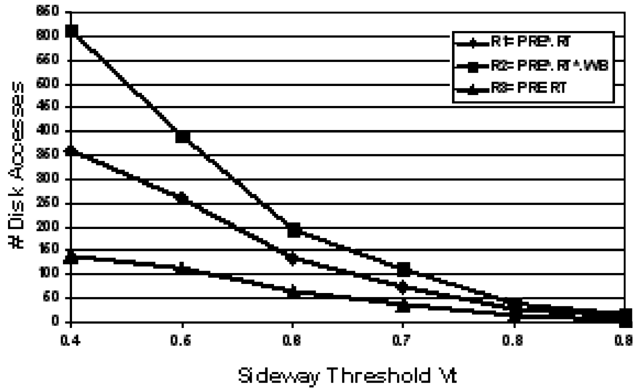
the same page; if there was not enough space in the current page then they were stored in the next page. HIndex to index hypernodes was initialized similarly.

A main memory-based sparse index was created to access any entry ⟨Tid, MType⟩ in *MIndex* (see Figure 14). In the sparse index, we divided ⟨Tid, MType⟩ entries into blocks (we used 1000 blocks). Each block corresponded to one or more pages in the *MIndex* file. The sparse index file contained the first ⟨Tid, MType⟩ in a given block and its physical address in *MIndex*. In order to retrieve all metalinks of type MType and emanate from Tid, we first used the sparse index to find the physical address of the first page with key ⟨Tid, MType⟩ in the *MIndex* file. If a given block in the sparse index corresponded to more than one page in the *MIndex* file then we may have needed to access more than one page in order to retrieve the metalinks for the specified key.

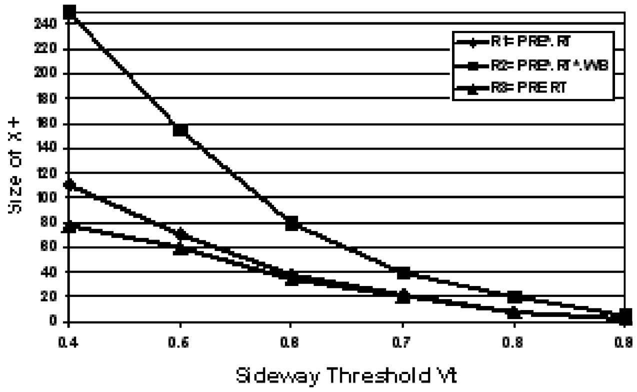
In the implementation, a disk-based metalink index *MIndex* with a page size of 1 kB was used to maintain all extracted metalinks. *MIndex* contained 2768 pages and had the size of 2.785 MB. We used a memory-based sparse index of size 1000; therefore, the first 768 blocks in the sparse index corresponded to three metalinks pages and the remaining 232 blocks corresponded to 2 pages. Thus, 1000 pages could be accessed using a single disk access; 1000 pages could be accessed using two disk accesses, and 768 pages required three disk accesses. In order to access the metalinks emanating from a given topic t, we needed a single disk access if topic t was in the first page in a given sparse index block, two disk accesses if it was in the second page, and three disk accesses if it was in the third page. Assuming that all pages contained the same number of metalinks and they were uniformly accessed, the expected average number of disk accesses (*avgDA*) to locate metalinks emanating from a given topic t was 1.92.

8.3 Experiments

In the experiments of this section, the behavior of TClosure algorithms was evaluated using different values for the regular expression, input topic size, sparse index size, and page size.



(a) Number of disk Accesses



(b) Size of the output topics X^+

Fig. 15. Threshold-based TClosure algorithm using different regular expressions.

8.3.1 *Regular Expressions.* We used three regular expressions, namely, $PRE^*.RT$, $PRE^*.RT^*.WB$, and $PRE.RT$ regular expressions to evaluate the performances of TClosure algorithms.

—*Observation* 1 (*Figure* 15 *and Figure* 16): Among the three regular expressions, the regular expression PRE.RT had the lowest number of disk accesses, and the smallest closure (i.e., X+) for both top-k and threshold-based TClosure algorithms.

—*Observation* 2 (*Figure* 15): For the threshold-based TClosure algorithm, the increase in both the number of disk accesses and the size of output topics X^+ was nonlinear with respect to the decrease in the sideway value threshold V_t. When the sideway importance value V_t was large then there was a small difference between the numbers of disk accesses using different regular expressions. But the difference became very large when V_t was small.

—*Observation* 3 (*Figure* 16): For all three regular expressions, the increase in the number of disk accesses was linear with respect to the increase in top-k topics for the top-k-based algorithm.

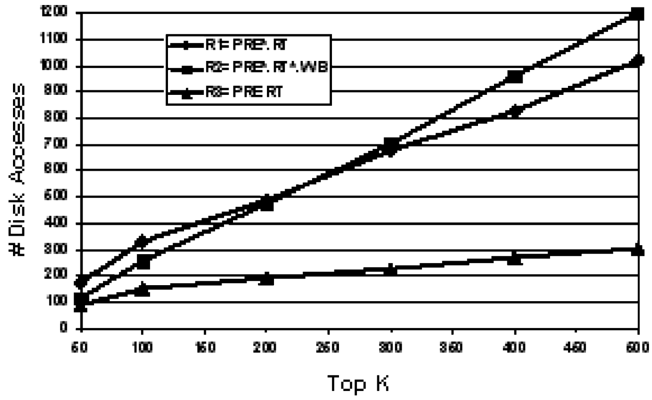


Fig. 16. Top-k-based TClosure algorithm using different regular expressions.

—*Observation* 4 (*Figure* 15): Among the three regular expressions, the regular expression *PRE**.*RT**.*WB* has the highest number of disk accesses and the largest closure (i.e., X^+) size for the threshold-based algorithm.

—*Observation* 5 (*Figure* 16): For the top-k algorithm, the regular expression *PRE**.*RT* had the highest number of disk accesses when k was less than 250. The reason for such a behavior is that the importance scores for the *WrittenBy* metalinks were 1.0, forcing the algorithm to locate topics with the highest importance using fewer disk accesses.

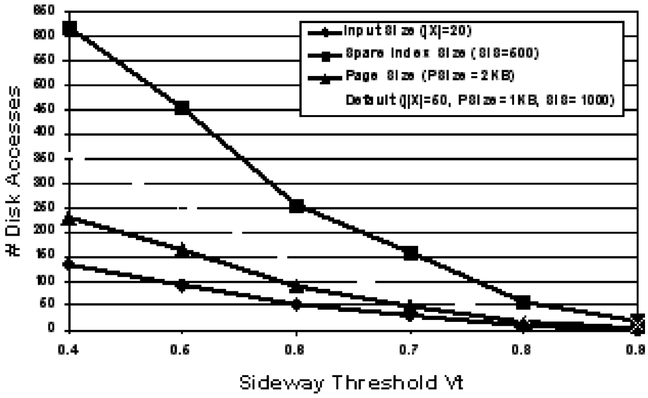
8.3.2 *Input Size, Page Size and Sparse Index Size.*

—*Observation* 6 (*Figures* 17 *and* 18): When the number of input topics decreased then both the number of disk accesses and the sizes of the output topics were decreased almost linearly.

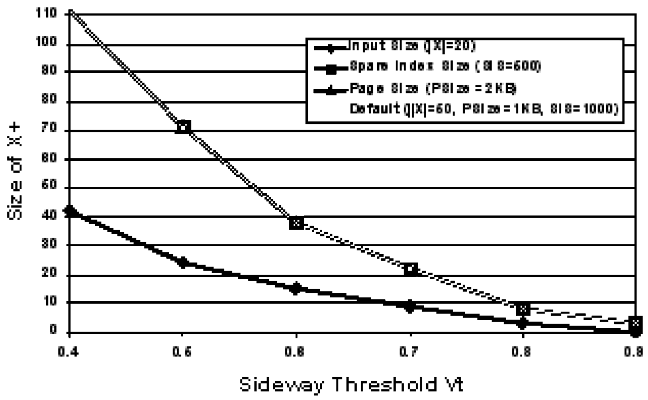
—*Observation* 7 (*Figures* 17 *and* 18): When the page size or sparse index size was changed then the number of disk accesses were changed with almost a constant rate for both top-k and threshold-based algorithms.

When the page size was increased from 1 to 2 kB then the number of disk-based pages in the *MIndex* file was decreased from 2768 to 1340 pages. Therefore, the expected number of disk accesses per requested metalink was decreased from 1.92 to 1.25 (1000 pages could be accessed using one disk access and 340 required two disk accesses). Thus, the expected number of disk accesses per traversed metalink was decreased by the ratio of 1.25/1.92 = 0.65. Figures 17 and 18 illustrate that the number of the disk accesses per metalink instance was decreased by the ratio of 0.55 to 0.67 for threshold-based algorithms and by the ratio of 0.62 to 0.65 for top-k algorithms.

When the size of the sparse index was reduced from 1000 to 500 blocks then the expected number of disk accesses per traversed metalink instance was changed from 1.92 to 3.29 (since there were 2768 pages; 500 pages required one disk access, 500 pages required two disk accesses, 500 pages required three disk accesses, 500 pages required four disk accesses, 500 pages required five disk accesses, and 268 pages require six disk accesses).



(a)Number of disk accesses



(b) Size of the output topics X^+

Fig. 17. Threshold TClosure algorithm using different values for the input size, page size, sparse index size.

Therefore, the expected rate became 3.29/1.92 = 1.7. As expected, and Figures 17 and 18 illustrate that the number of the disk accesses per requested metalink was increased by the ratio of 1.72 to 2.1 for threshold-based algorithms and by the ratio of 1.76 to 1.88 for top-k algorithms.

8.3.3 *Different Formulas for* Pre *Metalink Importance Scores.* We evaluated the performances of TClosure algorithms using different metalink importance score computations. We used the following two formulas to compute the importance scores of Prerequisite metalinks:

$(F1)\, \mathit{Imp}(\mathit{Pre}(\mathrm{P_i}, \mathrm{P_j})) = (\mathrm{O}(\mathrm{P_i}, \mathrm{P_j}) + 1)/(\mathrm{O_{max}}(\mathrm{P_j}) + 1)$

$(F2)\, \mathit{Imp}(\mathit{Pre}(\mathrm{P_i}, \mathrm{P_j})) = \mathit{MinPreFactor} + (1 - \mathit{MinPreFactor}) * \mathrm{O}(\mathrm{P_i}, \mathrm{P_j})/\mathrm{O_{max}}(\mathrm{P_j})$

where $0 \leq \mathit{MinPre} \leq 1$.

—*Observation* 8 (*Figures* 19 *and* 20): For both top-k and threshold-based TClosure algorithms, formula F2 with *MinPre* of 0.5 had the highest number of

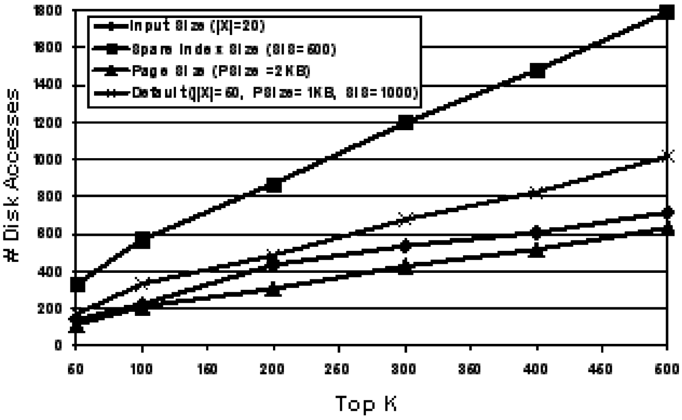


Fig. 18. Top-k TClosure algorithm using different values for the input size, page size, sparse index size.

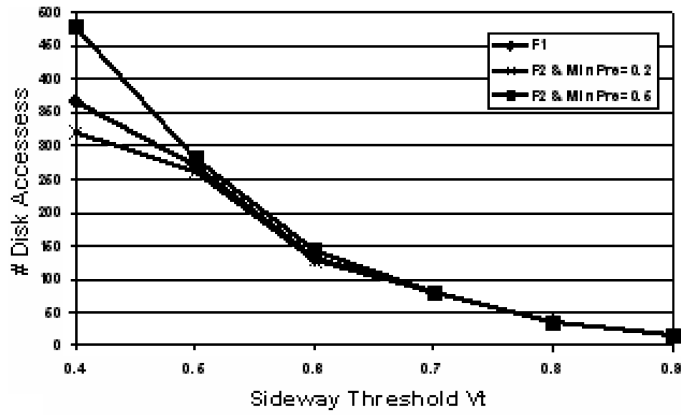
disk accesses and formula F2 with *MinPre* of 0.2 had the lowest number of disk accesses.

—*Observation* 9 (*Figures* 19 *and* 20): The differences between the number of disk accesses using different formulas were very small when the sideway threshold V_t was large (or when k was small).

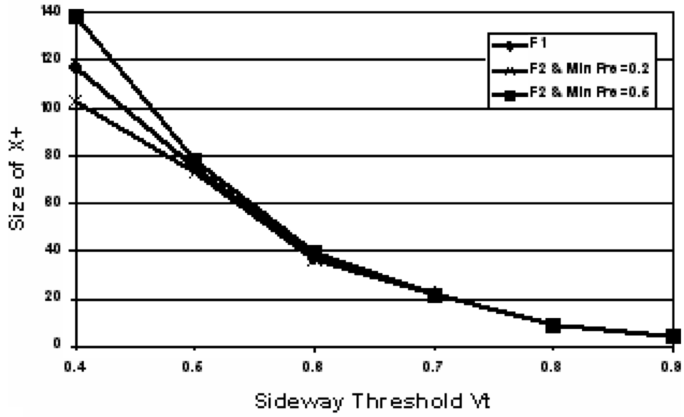
9. RELATED WORK

9.1 Web Data Extraction, Web Querying, and Web Metadata Models

Automatically extracting entities and relationships about entities from Web documents would be very useful for Web resource querying [Özsoyoğlu and Al-Hamdani 2003]. DIPRE [Brin 1998] employs a handful of training tuples of a structured relation R (which represents a specific meta-relationship among entities in the data) to extract all the tuples of R, from a set of HTML documents. DIPRE uses the training tuples to generate new patterns, and uses the newly generated patterns to extract more tuples, and so on. Snowball [Agichtein and Gravano 2000; Agichtein et al. 2000], an extension to DIPRE, improves the quality of the extracted data by including automatic patterns and tuple evaluation. One of the key improvements is that Snowball's patterns include named-entity tags. In addition, Snowball eliminates unreliable tuples and patterns by using strategies to estimate the reliability of the extracted tuples and patterns. The Proteus information extraction system [Grishman 1997; Grishman et al. 2002] divides the extracted text into sentences and into tokens, performs a lexical lookup for each token, and determines its parts-of-speech and features. Next, finite-state patterns are used to recognize names, nouns, verbs, and other special forms. Then the scenario pattern matching is used to extract events and relationships for a given relation. Proteus also uses an inference process to locate implicit information and make it explicit, and combines all the information about a single event using event emerging rules. The extracted events and phrases are used to update the database. QXtract [Agichtein and Gravano 2003]



(a) Number of disk accesses



(b) Size of the output topics X^+

Fig. 19. Threshold-based TClosure algorithm using different formulas for *Prerequisites* importance scores.

uses automated query-based techniques to retrieve documents that are useful for extracting a target relation from a large collection text documents. The field of (meta)data extraction from the Web, while promising, has a long way to go at this stage.

There have been a number of articles about querying the Web via a database-style query language; for a comprehensive survey, see Florescu et al. [1998]. Our work is distinguished from these works in that our focus is on (i) a metadata model for a Web resource (as opposed to the whole Web), and (ii) generic SQL extensions, and the associated query processing, for score management and text support. The SQL extensions, the associated query processing, and the proposed SVA operators are not necessarily restricted to metadata databases and Web querying; they can also be equally valuable for databases/applications dealing with score manipulations.

There have been extensive research and standardization efforts on information representation models for the Web. Two well-publicized metadata

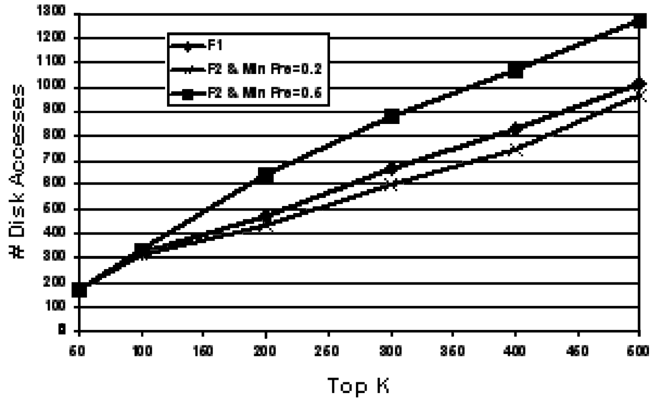


Fig. 20. Top-k TClosure algorithm using different formulas for *Prerequisites* importance scores.

standards for Web pages are Dublin Core and the Warwick framework. As summarized in Kobayashi and Takeda [2000], Dublin Core specifies a set of 15 metadata elements (e.g., title, creator, subject, etc.) for Web pages. More recent and more comprehensive proposals to add semantics to the Web include Topic Maps, the Resource Description Framework (RDF), and the Semantic Web effort. A Topic Map data model, as described in Biezunski et al. [1999], is similar to the Entity-Relationship model specialized for the abstract domain of topics and topic-related information. Our metadata model can be seen as a subset and an application of the Topic Map model, stripped of many details, stored in an object-relational DBMS, and enriched with the notion of importance scores. The Resource Description Framework [Lassila and Swick 1999] is a graph-based information model designed to describe Web information sources by attaching metadata specified in XML. In Lacher and Decker [2001], RDF and Topic Maps are shown to be equivalent in expressive power in that each is able to express the other. Semantic Web [Semantic Web; Berners-Lee 2000] is an RDF schema-based effort to define a semantic-based architecture for Web resources, with multiple layers that include a schema layer, a logical layer, and a query language. RQL [Karvounarakis et al. 2001] is a declarative language to query portal catalogs that are created according to the RDF standard in the context of the C-Web project. The RQL query engine attempts to optimize a query at the rewriting stage, and then leaves the job to the underlying ODBMS. In comparison, we propose a set of language extensions and evaluation algorithms that are integrated into the query engine. And we propose new operators for text similarity joins and topic closure.

9.2 Function Evaluation, Text Similarity Joins, and IR-Style Solutions

The notion of user-defined functions (UDF) has been around for quite a long time (i.e., SQL table functions [Reinwald and Pirahesh 1998], etc.), and can perhaps be used for application-based score management. In comparison, we propose a database-centric, native approach to score management: the SQL language and a query engine which, together, make use of input tuple scores in an *embedded* manner to answer queries, viewing scores as a native and internal property of a database schema. In this respect, our algebra combines function and score manipulation with traditional query processing in a new and unique way.

As a particular sort of score-generating predicates, we consider IR-style text similarity functions, which we assume to be natively embedded as *threshold predicates* in the system, as opposed to implementing them as user-defined functions. Today's commercial DBMSs provide full-text indexing and relevance ranking features for querying single text attributes (e.g., Oracle 9i Text [Oracle Corp. 2003], IBM DB2 Text Extender [IBM Corp. 2003], SQL Server 2000 Full-Text Search [Microsoft Corp. 2003]). In contrast, we allow similarity computations and comparisons not only as selection predicates, but also as join conditions. And, as mentioned before, the scores returned by SVA operators are employed during intermediate stages of query processing to limit the output space, and used to revise final output tuple scores dynamically; this has not been proposed in a commercial DBMS or a research prototype.

An earlier work that make use of text-similarity as a join condition was presented by Meng et al. [1998]. This work described three nested-loops-based algorithms to find top-k documents of a relation that are most similar to each document from another relation. These three algorithms are distinguished in their use of an inverted index, that is, the first algorithm directly compares document vectors from both relations, whereas the second one builds an inverted index for one of the relations, and the third one employs inverted indices for both of the relations. The underlying document representation model is the vector space model as used in our work. Our work differs from Meng et al. [1998] in that (a) our emphasis is on importance score handling, (b) our threshold predicates join tuples representing metadata (with relatively shorter text fields compared to entire documents), and (c) we make use of a maximal similarity filter as an early termination heuristic. Additionally, Meng et al. [1998] algorithms retrieve top-k (most similar) tuples for *each* tuple in the "other" relation whereas our top-k algorithms simply retrieve top-k (most similar) tuple pairs from the (implicit) Cartesian product of two relations in a global manner. Note that the inverted index-based approaches are also applicable to our similarity join algorithms; but Meng et al. [1998] reported that these approaches can only be efficient when one of the relations is very small (so that the index can fit into the main memory). In Section 7, we made use of an in-memory inverted index for the blocks of the outer relation (R) read into the memory during the nested-loops-based join processing.

Cohen [1998] described a new language, called *WHIRL*, that uses IR-based methods for similarity joins provided as built-in predicates in a data integration system. Our work has benefited from WHIRL, which also makes use of the maximal similarity heuristic (though in the context of the A* search algorithm proposed for query processing). However, our study emphasizes a general framework for handling scores during query processing, and threshold predicates in selection and join conditions are only one particular way of generating such scores, in addition to UDFs or other possible score-generating predicates.

More recently, database solutions that make use of IR techniques (and vice versa) have attracted research interest. A number of works have proposed allowing free-form keyword search over relational databases (e.g., DBXplorer [Agrawal et al. 2002], Discover [Hristidis and Papakonstantinou 2002], BANKS [Bhalotia et al. 2002; Hristidis et al. [2003]). These works fundamentally differ from ours in that they were intend to provide a free-form keyword search functionality over databases by automatically identifying and assembling (joining) a set of separate tuples that constitute a query answer as a whole. Other than relying on IR-based similarity computation techniques (employed for evaluating our threshold predicates), our work does not have many common points with the above-listed works. For instance, BANKS provides browsing and keyword search for online databases by modeling the database as a graph where nodes are tuples and edges are connections, such as the primary-foreign key relationships. An answer to a keyword query is a subset of this graph, which is modeled as a Steiner tree, with a set of nodes (tuples) including specified keywords and a central informative (root) node. These output tuple trees are also assigned scores according to node weights, edge weights, and the notion of prestige (similar to the famous Page-rank). Clearly, BANKS is not a competitive approach with respect to ours, but indeed can be complementary as it can operate on our metadata database just like any other ordinary database (possibly by turning off our extended SQL and using its own graph-based algorithms).

9.3 Ranked Query Evaluation

The topic of *top-k queries* has been the subject of extensive research recently. Carey and Kossmann [1997] introduced the *stop after* operator, which is an explicit and declarative way of restricting the cardinality of a query result in SQL. If the input stream is sorted, the *scan-stop* operator simply returns the first k tuples arriving as input (in a pipelined manner) and then closes down its input stream. In the case of unsorted input, the input stream must first be sorted to produce the top-k tuples. Our work is distinguished from Carey and Kossmann's [1997] work in that, instead of using a generic operator that simply reduces the output size of all other operators, SVA operators themselves are *aware* of the cardinality limitation (the SV threshold or the top-k value), and they only produce the requested tuples. SVA operators with top-k stopping conditions can be used in accordance with the conservative and aggressive strategies proposed by Carey and Kossmann [1997] (as top-k cannot propagate deeper in the operator tree safely). In this article, we adapted the conservative approach for defining our query semantics with top-k stopping condition. In a followup article Carey and Kossmann [1998], additional strategies were proposed for processing *stop after* queries. In contrast, SV threshold-based stopping conditions, which are unique to our work, safely propagate to all intermediate operators in the query tree (see Section 4). Thus, SVA operators with threshold-based stopping conditions can be used anywhere in place of their counterparts in relational algebra.

In a similar fashion to our SVA operators with top-k stopping conditions, top-k selection and join algorithms have been proposed. Two such works on top-k selection were Chaudhuri and Gravano [1999] and Chang and Hwang

[2002], and the latter also supported expensive predicates. We have discussed the processing of SVA selection operator elsewhere [Al-Hamdani and Özsoyoğlu 2003]. An early algorithm for top-k join was provided in Fagin [1999], and it was further optimized by Güntzer et al. [2000]. These algorithms assume equi-join conditions. More recently, join algorithms that support user-defined (arbitrary) join predicates have also been proposed, such as the J* algorithm [Natsev et al. 2001]. In comparison, we give nested-loops-based algorithms for top-k versions of the SVA join, and define a max filter heuristic for joins involving textual similarity (threshold) predicates. Our algorithms exploit score distributions and/or the similarity filter, and improve the performance considerably. Optimization of top-k predicates were also discussed in Mahalingam and Candan [2001], where the varying query outputs with respect to the different binding order of top-k predicates were taken into account.

Ranked-join operators by Ilyas et al. [2003, 2004] have similarities (and differences) with our work. In an earlier study [Ilyas et al. 2002], the authors proposed to encapsulate two previously existing rank join algorithms (namely, NRA and J*) in a physical join operator, with the focus of providing a ranked-join operator which can be used in pipelining query plans with join hierarchies. To this end, the NRA algorithm was modified to work in an incremental and pipelining manner. In a followup work [Ilyas et al. 2003], the authors proposed a new-rank join algorithm and two physical join operators that implement the new algorithm by using variants of the ripple join. Most recently [Ilyas et al. 2004], the authors introduced “interesting rank expressions,” extended dynamic programming-based query optimization to generate candidate plans that employ the rank-join operator, and proposed a probabilistic model to estimate the input cardinality (and subsequently the cost) of rank-join operators for query optimization purposes.

Both our work and the work of Ilyas et al. [2002, 2003, 2004] have concentrated on supporting score-aware operators in the query engines; however, the two approaches significantly differed in various aspects: first, we defined a general framework for a set of algebraic operators (namely, selection, join, and closure) which can (i) modify scores with newly introduced threshold predicates involving textual similarities, (ii) compute and propagate scores with respect to UDF predicates, and (iii) enforce stopping conditions based on either a threshold or a top-k constraint. For our extended-SQL queries, we have discussed the semantics of algebraic expressions involving our SVA operators interleaved with ordinary RA operators, and have shown that the proposed extensions are well-defined. In comparison, Ilyas et al. [2002, 2003, 2004] focused on defining a rank-join operator for pipelining query plans and optimization and cost evaluation issues for queries with a sequence of rank-join operators.

In comparing our SVA join operator and the rank-join operator of Ilyas et al. [2002] the most important distinction is our use of the threshold and UDF predicates, which *arbitrarily* change (increase or decrease) the scores of output tuples, making the results of Ilyas et al. not directly applicable to our SVA join algorithms. Put another way, output tuple scores of SVA join are dependent on *tuple component values* that are involved in score-modifying predicates, which is not the case in Ilyas et al.'s [2003] rank-join framework. In comparison, the rank-join [Ilyas et al. 2003] applies the *same* output score generation function and *only* to the scores of joining tuples. Another difference is that we allow the SVA operator itself to be aware of the top-k stopping condition (whenever allowed by our score-conservative policy) to reduce the intermediate output size in complex query trees. In contrast, in Ilyas et al.'s work, a Scan-Stop(k) [Carey and Kossmann 1997] operator is applied on top of the uppermost rank-join operator, and the join operators themselves do not know the top-k constraint. Having said this, adapting the physical join operators as proposed by Ilyas et al. for our SVA join algorithms is a future research direction.

9.4 Transitive Closure

SQL/TC is an extension to SQL to express generalized transitive closure queries [Dar and Agrawal 1993]. A directed graph G instance can be represented using a relation R with two columns S and T, where there is a tuple in R with values s and t for S and T if and only if there exists an edge from node s to node t in graph G. The transitive closure TC(G) of the graph G corresponds to the transitive closure TC of relation R with respect to S and T. Each edge in graph G has a value, and the value of an edge in TC(G) is derived from the values of the edges in the corresponding path-set. Dar et al. [1991] presented polynomial algorithms for transitive closure with restricted paths. SQL/TC has a complex syntax, and does not support computing the topic closure with top-k predicates, regular expressions, or hypernodes.

SQL'99 supports recursive queries using a "WITH RECURSIVE" statement [Eisenberg and Melton 1999; Lewis et al. 2003]. A recursive query is composed of two parts: the definition of a recursive relation and the query against the definition. The recursive queries employ a complex syntax to express the topic closure operator, and do not deal with closure with top-k predicates, regular expressions, and hypernodes.

9.5 Other Work

In our earlier work, we described the topic-based metadata model in more detail as well as some practical approaches for constructing such databases (e.g., the DBLP metadata database) [Altingövde et al. 2001; Özel et al. 2004]. This article extends our preliminary results for the SVA framework [Özsoyoğlu et al. 2002] as follows: first, SVA algebra operators are defined more completely, and illustrated with logical query tree examples. Second, threshold and UDF predicates for SQL are introduced. Third, the semantics of SQL extensions (correctness notion for "well-defined" queries) are defined, and proven correct. Last, but not least, complete experimental evaluations of the SVA join and topic closure are reported, for which the importance scores of topics and metalinks are computed from real world data, rather than synthetic data.

Very recently, Al-Khalifa et al. [2003] proposed a score-based framework for querying structured text in XML databases. This work also extended common algebraic operators and defines new ones for score manipulation; however, its focus was on providing IR-style ranked querying facilities for XML documents.

10. CONCLUSIONS

In this article, we have proposed a native score management and approximate text-similarity support to databases, to be used for Web resource querying on metadata extracted from the Web resource a priori. To this end, we have proposed SQL language extensions, algebraic extensions, and query processing algorithms that implement the proposed extensions.

Future work will include (i) adding new (e.g., "top-k") predicates to SQL extensions, and (ii) removing the closed-world assumption in a controlled manner, and adding focused crawler executions (at the Web information resource) during query evaluation time to those SVA operator evaluations that do not have "sufficiently large" number of output tuples.

An electronic appendix to the article is available in the ACM Digital Library.

REFERENCES

ACM SIGMOD ANTHOLOGY. Available online at `http://www.acm.org/sigmod/dblp/db/anthology.html`.

AGICHTEIN, E., ESKIN, E., AND GRAVANO, L. 2000. Combining strategies for extracting relations from text collections. In *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (Dallas, TX, May).

AGICHTEIN, E. AND GRAVANO, L. 2000. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the 5th ACM International Conference on Digital Libraries* (June).

AGICHTEIN, E. AND GRAVANO, L. 2003. Querying text databases for efficient information extraction, In *Proceedings of the 19th IEEE International Conference on Data Engineering* (ICDE; Bangalore, India, March).

AGRAWAL, S., CHAUDHURI. S., AND DAS, G. 2002. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering* (San Jose, CA, Feb.).

AL-HAMDANI, A. 2003. ACM anthology metadata extraction: Index and similarity factor construction. Tech. Rep. EECS Deptartment, Case Western, Reserve University, Cleveland, OH.

AL-HAMDANI, A. AND ÖZSOYOĞLU, G. 2003. Selecting topics for web resource discovery: Efficiency issues in a database approach. In *Proceedings of the DEXA Conference* (Prague, Czech Republic, Sep.).

AL-KHALIFA, S., YU, C., AND JAGADISH, H. V. 2003. Querying structured text in an XML database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Diego, CA, June).

ALTINGÖVDE, I. S., ÖZEL, S. A., ULUSOY, Ö., ÖZSOYOĞLU, G., AND ÖZSOYOĞLU, Z. M. 2001. Topic-centric querying of Web information resources. In *Proceedings of the DEXA Conference* (Munich, Germany, Sep.).

BERNERS-LEE, T. 2000. Semantic web roadmap. W3C draft. Available online at `http://www.w3.org/DesignIssues/Semantic.html`.

BHALOTIA, G., HULGERI, A., NAKHEY, C., CHAKRABARTI, S., AND SUDARSHAN, S. 2002. Keyword searching and browsing in databases using BANKS. In *Proceedings of the 18th IEEE International Conference on Data Engineering* (San Jose, CA, Feb.).

BIEZUNSKI, M., BRYAN, M., AND NEWCOMB, S. Eds. 1999. ISO/IEC 13250 Topic Maps. Available online at `http://www.ornl.gov/sgml/sc34/document/0058.htm`.

BRIN, S. 1998. Extracting patterns and relations from the World Wide Web. In *Proceedings of WebDB Workshop at EDBT* (Valencia, Spain, March). Available online at `http://citeseer.nj.nec.com/brin98extracting.html`.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst. 30*, 107–117. Available online at `http://citeseer.nj.nec.com/brin98anatomy.html`.

CAREY, M. J. AND KOSSMANN, D. 1997. On saying "Enough already!" in SQL. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Tucson, AZ, May).

CAREY, M. J. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *Proceedings of the 24th International Conference on Very Large Data Bases* (New York, NY, Aug.).

CHANG, K. C.-C. AND HWANG, S.-W. 2002. Minimal probing: Supporting expensive predicates for top-*k* queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Madison, WI, June).

CHAUDHURI, S. AND GRAVANO, L. 1999. Evaluating top-*k* selection queries. In *Proceedings of the 25th International Conference on Very Large Data Bases* (Edinburgh, Scotland, Sep.).

CHEN, L. 2001. Finding related papers in a digital library. Master's project. Case Western Reserve University, Cleveland, OH. Available online at `http://art.cwru.edu/NSF/chen.pdf`.

CITESEER. 2003. Estimated impact of publication venues in computer science. Available online at `http://citeseer.ist.psu.edu/impact.html`.

CODD, E. F. 1980. Data models in database management. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling* (Pingree Park, CO, June).

COHEN, W. W. 1998. Integration of heterogeneous databases based on textual similarity. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Seattle, WA, June).

DAR, S. AND AGRAWAL, R. 1993. Extending SQL with generalized transitive closure. *IEEE Trans. Knowl. Data Eng. 5*, 5 (Oct.), 799–812.

DAR, S., AGRAWAL, R., AND JAGADISH, H. V. 1991. Optimization of generalized transitive closure. In *Proceedings of the 7th International Conference on Data Engineering* (Kobe, Japan, April).

EISENBERG, A. AND MELTON, J. 1999. SQL:1999, formerly known as SQL3. *ACM SIGMOD Rec. 28*, 1, 13–138.

FAGIN, R. 1999. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci. 58*, 83–99. An extended abstract appears in ACM PODS 1996.

FLORESCU, D., LEVY, A., AND MENDELZON, A. 1998. Database techniques for the World-Wide Web: A survey. *ACM SIGMOD Rec. 27*, 3(Sep.), 59–74.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2, 73–169.

GRISHMAN, R. 1997. Information extraction: Techniques and challenges. In *Proceedings of the Summer School on Information Extraction (SCIE-97)*, Maria Teresa Pazienza, Ed. Springer-Verlag, Berlin, Germany.

GRISHMAN, R., HUTTUNEN, S., AND YANGARBER, R. 2002. Real-time event extraction for infectious disease outbreaks. In *Proceedings of Human Language Technology Conference* (HLT, San Diego, CA, March).

GÜNTZER, U., BALKE, W.-T., AND KIESSLING, W. 2000. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases* Cairo, Egypt, Sep.).

HRISTIDIS, V., GRAVANO, L., AND PAPAKONSTANTINOU, Y. 2003. Efficient IR-style keyword search over relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases* (Berlin, Germany, Sep.).

HRISTIDIS, V. AND PAPAKONSTANTINOU, Y. 2002. DISCOVER: Keyword search in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China, Aug.).

IBM CORP. 2003. Db2 text extender. Available online at `http://www-3.ibm.com/software/data/db2/extenders/textoverview/`.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2002. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China, Aug.).

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2003. Supporting top-*k* join queries in relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases* (Berlin, Germany, Sep.).

ILYAS, I. F., SHAH, R., AREF, W. G., VITTER, J. S., AND ELMAGARMID, A. K. 2004. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Paris, France, June).

KARVOUNARAKIS, G., CHRISTOPHIDES, V., PLEXOUSAKIS, D., AND ALEXAKI, S. 2001. Querying RDF descriptions for community Web portals. In *Proceedings of the 17ièmes Journees Bases de Donnees Avancees* (BDA'01, Agadir, Morocco, Oct. 29–Nov. 2). 133–144.

KESSLER, M. M. 1963. Bibliographic coupling between scientific papers. *Amer. Documentat. 14*, 10–25.

KLEINBERG, J. 1998. Authoritative sources in hyperlinked environments. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Mathematics* (San Francisco, CA, Jan.).

KLEINBERG, J. 1999. Authoritative sources in a hyperlinked environment. *J. ASSOC. Comput. Mach. 46*, 5, 604–632.

KOBAYASHI, M. AND TAKEDA K. 2000. Information retrieval on the Web. *ACM Comput. Surv. 32*, 2, 144–173.

LACHER, M. S. AND DECKER, S. 2001. On the Integration of Topic Maps and RDF data. In *Proceedings of the International Semantic Web Working Symposium* (Stanford University, Stanford, CA, July 30–Aug. 1).

LASSILA, O. AND SWICK, R. R. 1999. Resource description framework (RDF) model and syntax specification. W3C Recommendation. Available online at `http://www/w3.org/TR/REC-rdf-syntax`.

LEWIS, P., BERNSTEIN, A., AND KIFER, M. 2003. *Database and Transaction Processing*, Addison-Wesley, Reading, MA.

LEY, M. DBLP Bibliography. Available online at `http://www.acm.org/sigmod/dblp/db/index.html`.

LI, L. 2003. Metadata extraction: RelatedToPapers and its use in Web resource querying. Master's Thesis. EECS Department, Case Western Reserve University, Cleveland, OH.

LIBRARY. The Library of Congress, Washington, D.C. Available online at `http://www.loc.gov`.

MAHALINGAM, L. P. AND CANDAN, S. 2001. Query optimization in the presence of top-k predicates. In *Proceedings of the Multimedia Information Systems Conference* (Villa Orlandi, Capri, Italy, Nov.).

MENG, W., YU, C. T., WANG, W., AND RISHE, N. 1998. Performance analysis of three text-join algorithms. *IEEE Trans. Knowl. Data Eng. 10*, 3, 477–492.

MICROSOFT CORP. 2003. Microsoft SQL server 2000 full text search service. Available online at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/createdb/cm_fullad_3bs2.asp`.

NATSEV, A., CHANG, Y., SMITH, J., LI, C., AND VITTER, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases* (Rome, Italy, Sep.)

ORACLE CORP. 2003. Oracle 9i text. Available online at `http://www.oracle.com/ip/index.html?text_home.html`.

ÖZEL, S. A., ALTINGÖVDE, I. S., ULUSOY, Ö., ÖZSOYOĞLU, G., AND ÖZSOYOĞLU, Z. M. 2004. Metadata-based modeling of information resources on the Web. *J. Amer. Soc. Inform. Sci. Tech. 55*, 2, 97–110.

ÖZSOYOĞLU, G. AND AL-HAMDANI, A. 2003. WWW Web resource discovery : Past, present, and future. *Proceedings of the ISCIS Conference* (Antalya, Turkey, Oct.) (invited paper). Available online at `http://art.cwru.edu/`.

ÖZSOYOĞLU, G., AL-HAMDANI, A., ALTINGÖVDE, I. S., ÖZEL, S. A., ULUSOY, Ö., AND ÖZSOYOĞLU, Z. M. 2002. Sideway value algebra for object-relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China, Aug.)

ÖZSOYOĞLU, G., BALKIR, N. H., CORMODE, G., AND ÖZSOYOĞLU, Z. M. 2000. Electronic books in digital libraries. In *Proceedings of the IEEE Advances in Digital Libraries Conference* (Washington, D.C., May).

ÖZSOYOĞLU, G., BALKIR, N. H., ÖZSOYOĞLU, Z. M., AND CORMODE, G. 2004. On automated lesson construction from electronic textbooks. *IEEE Trans. Knowl. Data Eng. 16*, 3, 317–331. Available online at `http://art.cwru.edu`.

PORTER, M. F. 1980. An algorithm for suffix stripping. *Program 14*, 3, 130–137. Available online at `http://www.tartarus.org/~ martin/PorterStemmer`.

REINWALD, B. AND PIRAHESH, H. 1998. SQL open heterogeneous data access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Seattle, WA, June).

RAMAKRISHNAN, R. AND GEHRKE, J. 2000. *Database Management Systems*, McGraw-Hill, New York, NY.

SALTON, G. 1989. *Automatic Text Processing*. Addison-Wesley, Reading, MA.

SMALL, H. 1973. Co-citation in the scientific literature: A new measure of the relationship between two documents. *J. Amer. Soc. Inform. Sci. 24*, 4, 28–31.

SEMANTIC WEB. The Semantic Web community portal. Available online at `http://www.semanticweb.org`.

Received January 2003; revised November 2003, May 2004; accepted June 2004